Automatic Generation of Test Cases for Agile using Natural Language Processing

Prerana Pradeepkumar Rane

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Thomas L. Martin, Chair
Steve R. Harrison
A. Lynn Abbott

March 14, 2017
Blacksburg, Virginia

Keywords: Automatic Test Case Generation, Agile Testing, Requirements Engineering

Automatic Generation of Test Cases for Agile using Natural Language Processing

Prerana Pradeepkumar Rane

ABSTRACT

Test case design and generation is a tedious manual process that requires 40-70% of the software test life cycle. The test cases written manually by inexperienced testers may not offer a complete coverage of the requirements. Frequent changes in requirements reduce the reusability of the manually written test cases costing more time and effort. Most projects in the industry follow a Behavior-Driven software development approach to capturing requirements from the business stakeholders through user stories written in natural language. Instead of writing test cases manually, this thesis investigates a practical solution for automatically generating test cases within an Agile software development workflow using natural language-based user stories and acceptance criteria. However, the information provided by the user story is insufficient to create test cases using natural language processing (NLP), so we have introduced two new input parameters, Test Scenario Description and Dictionary, to improve the test case generation process. To establish the feasibility, we developed a tool that uses NLP techniques to generate functional test cases from the free-form test scenario description automatically. The tool reduces the effort required to create the test cases while improving the test coverage and quality of the test suite. Results from the feasibility study are presented in this thesis.

Automatic Generation of Test Cases for Agile using Natural Language Processing

Prerana Pradeepkumar Rane

GENERAL AUDIENCE ABSTRACT

Testing is a crucial part of the software life cycle which ensures that the developed product meets the needs of the business. The Agile methodology is an iterative and incremental approach to software development. This work is a contribution to the testing process in Agile software development. Currently, test cases are written manually based on the requirements provided by the business. Manual test case generation is a time-consuming process. The business frequently changes the requirements thus wasting time and effort spent in manually writing the test cases. Also, not all the testers have prior knowledge of the working of the system which leads to difficulties in designing the test cases to match the requirements. In this work, we have developed a tool which can generate test cases suitable for Agile software development from the requirements provided. This automatic test case generation tool reduces the effort needed, improves the quality test cases and the coverage of the requirements by the generated test cases. This work can find application in industries that use Agile methodologies for developing and testing their products.

Acknowledgments

I am indebted to many people for the completion of this work:

- To Dr. Tom Martin for providing the opportunity to work on this project and his constant guidance starting with the wearable computing class to the completion of this thesis.
- To Professor Steve Harrison and Dr. Lynn Abbott for their valuable insights and feedback and for serving on my thesis committee.
- To the participants of the user study for their patience in enduring the 3-hour long process.
- To my parents, Pradeep and Nilima, without whom none of this would have been possible.
- To Akchat for encouraging me to achieve my goals.
- To my friends in the E-Textiles Lab, especially Jason and Kristen for their helpful suggestions.
- Finally, to all my friends who have always motivated me - Vishal, Arjun, Nikhitha, Divya, Clint, and Shravya.

Contents

List of Figures **viii**

List of Tables **xi**

1 Introduction **1**

1.1 Overview 1

1.2 Motivation 2

1.3 Problem Statement 3

1.4 Proposed Solution 3

1.5 Contribution 4

1.6 Thesis Organization 5

2 Background **6**

2.1 Agile Software Development 6

2.1.1 Evolution of Agile 6

2.1.2 Unified Modeling Language 8

2.1.3 Behavior Driven Development 8

2.2 Natural Language Processing 10

2.2.1 Stanford CoreNLP 10

2.2.2 WordNet 12

2.3 Grammatical Knowledge Patterns 12

2.4 Generation of Activity Diagrams and Test Cases 15

2.4.1 Generation of activity diagrams from natural language requirements . 16
2.4.2 Generation of test cases from activity diagrams 16
2.4.3 Generation of test cases from Agile user stories 17
2.5 Research Contributions 17

3 Design & Implementation of Test Case Generation Tool 19
3.1 Design Goals 20
3.2 Design Constraints 20
3.3 System Design 21
3.4 Graphical User Interface 22
3.4.1 Features of the GUI 22
3.4.2 Using the Test Case Generation Tool 28
3.5 Stanford Parser 29
3.5.1 Lexical Analysis 29
3.5.2 Syntactic Analysis 30
3.6 Frame Structure Representation 31
3.6.1 Frame structure 31
3.7 Test Case Generation 34
3.7.1 Activity Diagram 34
3.7.2 Activity Graph 36
3.7.3 Depth First Search Algorithm 37

4 Evaluation of Test Case Generation Tool 38
4.1 Preliminary Analysis of Test Cases and Model 39
4.1.1 Test Coverage Criteria 39
4.1.2 Effort required to generate the test cases 39
4.1.3 Test Case Productivity 40
4.2 Developing a User Study 40
4.2.1 Feasibility Study Design 40

4.2.2 Study Description 44
4.3 Data Analysis Techniques 45
4.3.1 Correlation 45
4.3.2 Analysis of Variance 46
4.3.3 Hypothesis Testing 46

5 Results and Discussion **47**
5.1 Preliminary Analysis 47
5.1.1 Analysis of Quantitative Data 49
5.1.2 Discussion of Results 51
5.2 Feasibility Study 51
5.2.1 Analysis of Quantitative Data 52
5.2.2 Analysis of Qualitative Data 65
5.2.3 Discussion of Results 68
5.3 Findings of the Preliminary Analysis & Feasibility Study 68

6 Conclusion and Future Work **70**
6.1 Contributions of Automatic Test Case Generation Tool 70
6.2 Future Work for Automatic Test Case Generation 70

Bibliography **72**

A Installation Manual **76**

B IRB Approval **78**

C Survey for Feasibility Study **80**

D Data sheet for test cases generated manually and automatically **85**

List of Figures

2.1 Penn Treebank description of the POS Tags 13
2.2 Penn Treebank description of Phrase Tags 13
2.3 Different processes involved in test case generation 15

3.1 High-Level System Design 21
3.2 Automatic test generation tool 23
3.3 Dictionary 24
3.4 Retrieval from Database 25
3.5 Mapping of Test Scenario Description to User Stories 27
3.6 Error Handling messages 28
3.7 Natural Language Processing Block 29
3.8 Output of the POS Tagger 30
3.9 Output of the Dependency Parser - Phrase Structure Tree 31
3.10 Output of the Dependency Parser - Typed Dependencies 31
3.11 Constructs used in an Activity Diagram 34
3.12 Activity Diagram for the Online Shopping example 35
3.13 Activity Graph for the Online Shopping example 36
3.14 Test paths and test cases for the Online Shopping example 37

4.1 AstroNav Application 41
4.2 FitTrain Application 42
4.3 Mooditation Application 43

5.1 Activity Diagram for ATM cash withdrawal example 48
5.2 Activity Diagram for ATM cash withdrawal example using our tool 49
5.3 Test coverage criteria for ATM Cash Withdrawal case study 49
5.4 Test cases generated by the tool for ATM Cash Withdrawal case study . . . 50
5.5 Comparison of Test cases for ATM Cash Withdrawal case study 50
5.6 List of features 52
5.7 Manually generated test cases - Averages of parameters 52
5.8 Automatically generated test cases - Averages of parameters 52
5.9 Total time taken by each user 53
5.10 Total time taken for each feature 53
5.11 Breakdown of time taken in automatic method 54
5.12 Time taken per user Vs number of test cases generated by each user 54
5.13 Effort taken by each user per test case 55
5.14 Effort taken per feature 55
5.15 TCP by each user 56
5.16 TCP per feature 56
5.17 Total Coverage by each user 57
5.18 Total Coverage per feature 58
5.19 Activity diagram for Feature 3 by User 3 (manual) 58
5.20 Activity diagram for Feature 3 by User 6 (automatic) 59
5.21 Correlation 60
5.22 ANOVA between number of dictionary parameters and number of test steps 60
5.23 Hypothesis testing for time taken 61
5.24 Hypothesis testing for effort required 62
5.25 Hypothesis testing for Test Case Productivity 63
5.26 Hypothesis testing for Test Coverage of Requirements 64
5.27 List of sub-features 64
5.28 Time and effort for sub-features 65

5.29 User preference - Manual Vs Automatic 66
5.30 Responses to survey questions 66

List of Tables

3.1 Frame Structure for Active Voice. 32
3.2 Frame Structure for Passive Voice. 32
3.3 Frame Structure for Preposition. 32
3.4 Frame Structure for Conjunction. 33
3.5 Frame Structure for Precondition. 33
3.6 Frame Structure for Marker. 33

Chapter 1

Introduction

1.1 Overview

The aim of this thesis is to establish the feasibility of automatically generating test cases within an Agile software development workflow using natural language processing (NLP). The goal is to create functional test cases from business requirements. In Agile, business requirements are captured in the form of user stories. We will use existing natural language processing techniques to process the user stories into activity diagrams, which are used to generate test cases.

Agile software development [1] methods were designed to keep up with the rapid changes in the market, requirements of the business stakeholders and the needs of the customers. The traditional waterfall method for software development was replaced by Agile software development since the existing methodologies showed a significant failure rate in the industry. The purpose of Agile is to speed development while keeping the product aligned with the business expectations, by having development and testing go hand-in-hand from the start. Testing is continuously integrated into Agile from early developmental stages to ensure defect-free continuous deployment and that all requirements are met. Providing higher requirement coverage and defect detection through testing is largely dependent on test case design and execution.

The two development approaches that are widely prevalent are Test-Driven Development (TDD) and Behavior-Driven Development (BDD). Test-Driven Development (TDD) is an approach for developing software by writing test cases before writing functional code [2]. Behavior-Driven Development (BDD) was created by Dan North [3] to overcome the limitations of TDD since TDD was considered too unstructured. BDD uses "ubiquitous language" to describe the behavior of the system using smaller client-valued functions called *features*. BDD provides the pre-defined template for a *user story*, which is the feature or requirement to be implemented [4]. However, both these methods were developed before automated

testing and are unable to deliver a complete testing process.

Traditionally test cases were generated manually from the source code of the program or specifications during the post software development phase. Now, user requirements are most often captured using the natural language format since it is easily understandable and used by humans naturally. However, the use of natural language can also give rise to many errors in the development stage. To minimize these errors, requirements are transformed from natural language into computational models using Unified Modeling Language (UML), a standard medium for representing object-oriented designs. Model-Based Testing allows for the generation of test cases during the early development stages so that developers can identify inconsistencies and improve the specifications thus saving time and money. The UML Activity Diagram describes the behavior of the system using the flow of operations in a process [5]. In our work, we have developed a tool to derive test cases from natural language requirements automatically by creating UML activity diagrams.

1.2 Motivation

Previous work by Forsyth [6] had investigated the possibility of using electronic storyboarding in a design process to generate representative state diagram models and pseudocode. But through interactions with potential clients of their storyboarding tool, Forsyth [6] discovered that the clients were not interested in code generation. Instead, the clients were interested in the possibility of capturing design requirements in a behavioral model that could potentially reduce rework due to miscommunication between the product owner and developer/tester. Even though the clients were not keen on generating code for their designs, Forsyth [6] had the right idea of using models to create artifacts that could potentially supplement the design and development process.

In our work, we expanded on the idea of using UML models in Agile software development to create test cases. Through an informal discussion with some business analysts, we gained an understanding of the Agile processes followed in the industry and their shortcomings. We found that apart from modeling the requirements to facilitate understanding of changes in requirement, there was scope for improving the test generation methods. Every time there is a change in requirements, new test cases will need to be generated, and the tester will need to identify components that do not need to be re-tested. Hence we designed our tool to combine the easy representation of requirements using UML models and test case design based on the requirements and models.

Testing is one of the most crucial activities in the software development life cycle, consuming about 50% of the time and budget of the entire project [7]. Testing ensures that the system does what it is supposed to do in every single instance. Testing comprises of two parts: finding all the paths to be tested in the system and identifying relevant data to verify the system behavior. The test cases must be able to provide maximum coverage of paths and

data. Another issue negatively impacting the quality of tests, especially in Agile environments, is frequent changes in requirements. Additionally, requirements must be sufficiently detailed to create better quality test cases. Hence, there is a need for an automated test case generation tool which can help in reducing the cost of testing while providing maximum coverage.

1.3 Problem Statement

Creating test cases is a challenging and time-consuming aspect of software testing. The software testing process consists of test case generation, test execution, and test evaluation. While test execution and evaluation are relatively easier to automate, generating and deciding which test cases are sufficient to verify the system is more difficult. Test case generation covers 40-70% of the software test life cycle [8]. Current practices involve manually writing the test cases based on the functional requirements provided. Not all the testers have prior knowledge of the working of the system which leads to difficulties in designing the test cases to match the product owner's needs [9]. There is a considerable amount of communication required between the developer and the tester. This thesis investigates the possibility of a solution to overcome the limitations of the current practices. Autogeneration of test cases will help save time and money, improve the quality of testing and ensure better test coverage. Autogenerated tests will also facilitate easier maintenance and reuse of test cases if the customer changes the requirements. The following questions are addressed and supported by the contributions of this thesis.

1. Can we generate test cases from user stories in an Agile software development workflow using the automatic test case generation tool that we have developed?
2. Does this tool save the tester's time and effort while improving the quality and coverage of the test cases?

1.4 Proposed Solution

The proposed solution is to create a tool that can capture user stories as an input and produce functional test cases as an output. User stories are provided in Agile development in the format "As a [**role**], I want [**functionality**], so that [**business value**]." These user stories will then be processed using natural language processing techniques. A key objective of the tool is to minimize the amount of tedious work for the tester, both in original test case generation and future maintenance, and staying as close as possible to the existing process. The user stories will be supplemented with a *test scenario description* which will help in the generation of the test cases. The test scenario description contains relevant descriptive

information about the requirement to be built, permitting the generation of test cases from user stories with minimal human intervention.

Well written documentation is rare in Agile [10]. Most of the requirements are gathered through discussion with product owners. Agile also supports exploratory testing which relies on the tester's experience to find the flaws in the design or code. The usage of test scenario description fits well into the exploratory testing process since the test scenario is simply the sequence of interactions between the user and the system. Information (such as test scenario descriptions) that is common to similar user stories must be specified once and can be shared across user stories. Retaining shared information simplifies the generation of test cases by eliminating redundancy.

This thesis focuses on generating test cases from Agile user stories. First, we process the input in the form of a user story and test scenario description using a natural language parser. The natural language parser performs sentence splitting, part-of-speech tagging, and syntactic parsing to identify the relationship between the words. Using the dependencies derived from the parser, we create a UML activity diagram to show the flow of the functionality. From the activity diagram, we create the *activity graph* consisting of all the activities represented as nodes. The activity graph is traversed from the initial activity state to the final activity state to find all possible valid paths using the Depth First Search algorithm. The test paths are then used to generate all possible functional test cases.

1.5 Contribution

The main contribution of this thesis is a novel tool that uses Agile user stories to automatically generate test cases and create behavioral models using natural language processing. The tool can be used by testers who may or may not be experienced, to achieve the following improvements in testing within Agile software development. The tool:

1. reduces the time taken to create test cases.
2. reduces the effort (time per test case) required to create test cases.
3. improves the quality of test cases.
4. improves the test coverage of the requirements.
5. generates test cases for multiple user stories of a feature, further reducing the time and effort for the tester.

The utility and efficiency of the tool are verified through a user study. The participants of the user study evaluate an application to develop test cases manually and using the tool. The key parameters of evaluation are time and effort taken to generate the test cases. The

quality of the test cases is verified through the inclusion of boundary testing corner cases and minimal redundant test cases. The generated test cases are examined for completeness and accuracy using test coverage criteria, such as path and predicate coverage. As will be shown in Chapter 5, the tool increases the time to create test cases by 7% but reduces the time taken per test case by 31% and improves the test coverage of requirements by 23%.

1.6 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses the key underlying processes in this thesis such as Agile software development and testing. It also includes previous work involving the auto-generation of test cases in Agile software development that has influenced the creation of the tool. Chapter 3 provides a detailed design and implementation of the tool. Chapter 4 describes the feasibility study created to evaluate the system from a usability, accuracy and performance perspective. Chapter 5 will present the results of the feasibility study. Finally, Chapter 6 concludes the work and presents future directions this work can take.

Chapter 2

Background

This chapter describes the necessary information about Agile related processes and definitions, unified modeling language, and natural language processing techniques used in our work to understand our design and implementation decisions in Chapter 3, Design and Implementation of Test Case Generation Tool. Next, we will also discuss existing solutions which have influenced our design, and how the system presented in this thesis is different from other related work.

2.1 Agile Software Development

2.1.1 Evolution of Agile

Traditional software development models such as Waterfall model were widely used in the industry in the 1990s [11]. The Waterfall model consisted of the analysis phase in which a business analyst generally worked with the client to create a set of requirements and a model of the final product. Next, in the requirement specification phase, a requirements document was maintained that was the basis for Design, Development, Testing and release of the product. During the testing phase defects were raised if there was a mismatch in the requirements and developed code. The client was not involved in the development phase and would view the product after the Testing phase. If the client made any changes in the requirement, there was rework involved. The constant change in requirements and rework increases the time, cost and required resources for software development and could also lead to failure of the project [12, 13].

Modern software development techniques were developed to keep up with the increasing complexity of software and frequent changes in requirements. One such methodology was Agile. The Agile Alliance [1] created Agile Software Development in 2001. Agile is a set of frameworks that follows an iterative approach to developing products incrementally, keeping

the stakeholders in the loop, from the beginning of the project to the delivery of the product. Agile breaks down larger functionality into smaller pieces called user stories which are delivered in short two week cycles called sprints. The product owner is an important member of the Agile process who guides the team towards creating products that meet the customer's needs. The product owner keeps track of the features requested by the clients and breaks them down into user stories. The product owner also sets the acceptance criteria for the user story. The Agile approach, especially the user stories have a lot of advantages [11]: they replaced time-consuming documentation and encouraged face-to-face communication between developers and business owners which would lead to better understanding of requirements, decrease in misunderstanding. Agile also allows easier testing of deliverables and a faster feedback mechanism through the use of acceptance criteria.

Agile project management tools such as Rally, VersionOne, and Atlassian JIRA are being used in the industry to implement Agile software development methods such as Kanban and SCRUM for sprint planning, daily scrums, backlog management, and estimation of story points. In Agile development, and testing goes hand-in-hand. To support planning and execution of acceptance and regression testing, some tools such as VersionOne and Rally have their own test management suite [14]. JIRA integrates with test management plug-ins such as Zephyr, HP Quality Center to create and track test issues or integrate with test automation tools such as Selenium [15]. In Agile methodologies, a user story is marked complete if it passes all related acceptance tests. Acceptance tests are written by a member of the development team. Once a user story is completed and delivered, the testing team evaluates if the tests should be retained for future regression testing. Regression Testing ensures that newly developed additions to the system do not break the existing system. Testing members create the test plan, manage the test cases and the testing activities within a sprint.

Test case creation follows one of the following techniques:

1. Requirement-Based Testing

   Test cases are derived from the requirements which specify the inputs to the system, functionality being evaluated and the expected outcome. The key advantages of this technique are that the requirements define the significant behavior of the system and can be used as a basis for developing functional test cases [16].

2. Model-Based Testing

   Test cases are derived from the model of the system being developed. The advantages of this technique are that discrepancies are caught in the initial stages, reducing testing time. It also helps in finding inconsistencies in the requirements [17].

3. Code-Based Testing

   Test cases evaluate the code to ensure that the different test paths of the system are covered. The main benefit of this technique is that all untested parts of the software

are caught [18].

Our implementation aims to fit in this test management process to simplify the task of test case creation. Most of the existing research follows one of the techniques mentioned above to create test cases. In our implementation, we are generating test cases from the requirements using models.

2.1.2 Unified Modeling Language

Unified Modeling Language is often used with Agile development for modeling business applications using consistent notations [1]. Unified Modeling Language (UML) was developed by the Object Management Group in 2003 as the industry standard for developing software using Object-Oriented technology [1]. UML facilitates visualization of the behavior and interaction of objects from early developmental stages. UML helps the end users and the developers in viewing and maintaining the consistency between design and specification. Since UMLs growing popularity in modeling object-oriented software, there has been a lot of research on using UML in software testing [19–22]. Model-Based Testing makes use of UML to create and evaluate complex representation of the system. UML uses an object-oriented approach to model the user requirements. UML consists of static diagrams which represent the components of the system, behavior diagrams which model the behavior of the system and implementation diagrams which represent the flow of control among elements of the system [23]. The dynamic diagrams consist of Activity Diagram, State Diagram, and Use Case Diagram.

Activity Diagrams are best suited to describe the high-level business flow and low-level process flow. In this work, we will focus on representing the system using Activity Diagrams. The activity diagram is depicted in the form of a flowchart of activities and transitions between the activities. Edges represent the transition between activities. Each activity represents the behavior of a part of the system. An activity diagram can represent concurrent and sequential flows of control between activities. A decision node represents conditional activities. As shown in Chapter 3, we will use the activity diagram to understand the behavior of the system in terms of activities. These activities will be used to generate the activity graph and test cases.

2.1.3 Behavior Driven Development

The implementation described in Chapter 3 assumes a Behaviour Driven Development (BDD) approach to accepting the user requirements and generating the test cases based on the requirements. Before we dive into the specifics of BDD, we will understand why BDD was developed. Since 2003, Agile followed Test-Driven Development (TDD), an iterative process in Agile development where test code is written before the implementation code [1].

TDD improves on the drawbacks of traditional software development by incorporating testability into the design of the software. Acceptance-Test Driven Development (ATDD) is a variant of TDD which provides acceptance tests for the high-level system specifications [1]. TDD tests if the software is producing the desired output, while ATDD tests if the software matches the functionality you want. Despite the many advantages of TDD, it also has some drawbacks. Effective use of TDD requires practice and skill to understand "where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails" [3]. To overcome the limitations of TDD, Dan North [3] introduced the concept of Behaviour-Driven Development (BDD) in 2006 which combines the concepts of TDD and ATDD. The main feature of BDD is that it expresses the behavior of the software in natural language. BDD, using natural language as a medium, improves communication between all stakeholders of the project.

User Stories

BDD describes the natural language description of the high-level requirements of the system as *features*. Since the features capture high-level behavior, they can be too large to complete in a single iteration. Hence, they are broken into smaller subsets called *user stories*. The template [3, 4] for a story, as shown below, extracts information about the business value that a behavior will provide, for a user role.

As a - [user role]
I want - [functionality]
In order to - [provide business value]

Acceptance Criteria

Stories can be specified in different contexts called scenarios which describe the different behavior of the user story called *acceptance criteria*. The scenario describes the context of the story, the event that occurs and the outcome which is the acceptance criteria [3, 4], as shown below.

Given - context
When - action
Then - outcome

Existing BDD frameworks such as Cucumber and RSpec focus on the generation of low-level acceptance test code based on the natural language description. In our work, we are using BDD to capture user stories as an input to the system for functional test case generation.

Acceptance criteria are used to indicate the expected results. For the purpose of our system, we are not using the word "scenario" in the context defined by BDD. Instead, we are using "test scenario description," explained in Chapter 3.

2.2 Natural Language Processing

The primary input to the tool are requirements expressed in natural language. Hence Natural Language Processing (NLP) techniques play a vital role in our implementation. Existing NLP techniques such as POS Tagging, Dependency Parsing, Lemmatization, and Synonym Generation are used. This thesis uses NLP to match the requirements to a relevant test scenario description in the system. The system also searches for synonyms if an exact match between the test scenario description and user story is not found. The requirements are then parsed to identify the part-of-speech tags and typed dependencies between the words and extract relevant information using frame structures. More details about the role of NLP techniques in the implementation process will be described in Chapter 3. This section discusses the Stanford CoreNLP which is used for identification of POS tags, lemma and dependencies; and WordNet which helps in recognizing synonyms of a word.

2.2.1 Stanford CoreNLP

Stanford CoreNLP is an open source Java-based suite of natural language processing tools, initially developed in 2006 by the Stanford Natural Language Processing (NLP) Group [24]. It has numerous functions ranging from Part-Of-Speech (POS) tagging, analyzing sentiments to providing syntactic analysis of a sentence. The tool uses several features of the Stanford CoreNLP suite such as the tokenizer, sentence splitter, POS tagger and dependency parser. A natural language parser deciphers the grammatical structure of sentences using probabilistic knowledge of the language. Probabilistic parsers produce the most likely analysis of sentences and perform better than statistical parsers [25]. The Stanford NLP parser implements a probabilistic natural language parser based on Probabilistic Context-Free Grammars (PCFG) and outputs Universal Dependencies and Stanford Dependencies. The underlying assumption of dependency parsing is the idea that grammatical structure consists of words linked by binary asymmetrical relations called dependency or grammatical relations. These dependencies represent the relationship between the words of a sentence and help in extracting relevant information from a complex sentence.

Stanford Representation

The Stanford typed dependencies representation [26] extracts textual relations based on the grammatical structure of a sentence. The dependencies are grammatical relations between

a governor and a dependent. The governor and dependent are words in the sentence from which the grammatical relation is extracted, appended by the word number. The format for the dependencies is *abbreviated_relation_name*(governor, dependent). Stanford parser provides four variations of typed dependency representations - basic, collapsed, propagation and collapsed tree [27]. The main difference between the typed dependencies is the manner of representation - in a tree form, or in cyclic graph form. The dependencies are:

1. Basic typed dependencies use a tree structure with each word in the sentence being a dependent of another word.
2. Collapsed dependencies include cyclic dependencies such as relative clause modifiers (rcmod) and controlling subject (xsubj) which may break the tree structure, turning the structure into a directed graph. Collapsed dependencies also simplify the relations involving prepositions and conjuncts to get direct relations between their words.
3. Propagation dependencies are a form of collapsed dependencies which propagates subject and object relations to the words of the conjunct (conj) dependencies. This is the default representation.
4. Collapsed tree dependencies are a form of collapsed dependencies which exclude the relations which break the tree structure. This format does not allow the propagation of conjunct relation or the processing of relative clause, but it supports the simplification of relations.

Universal Representation

Since its original conception as the backend support for Stanford parser in 2005, Stanford dependencies have evolved as a widely used standard for dependency analysis. Similarly, Google universal tags have been accepted as a widely used standard for mapping part-of-speech tagsets to a common standard [28]. In 2013, an effort was made to combine Stanford dependencies and the Google universal tags into an annotation scheme called the Universal Dependency Treebank [28]. The new Universal Dependencies created in 2014 is a single framework designed to add or improve the defined syntactic relations to better accommodate different grammatical structures in various languages [29]. The current version of Stanford typed dependencies contains 50 grammatical relations which make use of the Penn Treebank POS and phrase tags. Since version 3.5.2 of the Stanford Core NLP, Universal dependencies is the default representation for extracting grammatical relations. Universal dependencies offer the following representations [30] of typed dependency relations between words:

1. Basic dependencies represent typed dependencies in a tree structure with an only word dependent on the root, as the head of the sentence.

2. Enhanced dependencies improve on the basic dependencies by augmenting modifiers and conjuncts and making implicit relations more explicit.

3. Enhanced++ dependencies overcome the shortcomings of the earlier versions especially on multi-word prepositions and light noun constructions.

In this thesis, we will not be focusing on the differences between the representations or their performance. We will be using the Stanford Parser's latest default representation, i.e., the Universal Representation with Enhanced++ dependencies.

Treebank

A treebank is a set of words represented in a tree form annotated with syntactic information. Treebanks perform annotation based on the phrase structure of a sentence and the dependencies between words. Similar to the POS tags, phrase tags are assigned to a group of co-located words in a sentence (phrase).

The Stanford Parser uses the Penn Treebank for annotating the sentences with parts-of-speech tagsets. The Penn Treebank is a human-annotated collection of 4.5 million words [31] which groups elements using phrase tags and POS tags in a phrase tree structure. The dependency structure representation considers the other words in the sentence to be directly or indirectly independent on the verb of the sentence. Figures 2.1 and 2.2 provide a list of the POS tags, Phrase tags and the description of the tags based on the Penn Treebank project [32].

2.2.2 WordNet

WordNet [33, 34] is a lexical database developed by the Princeton University. It combines different parts of speech (nouns, verbs, adjectives, and adverbs) from the English language into logically similar sets called synsets. Synsets link groups of words based on not just their lexical relations but also the meaning or sense of the phrase. It contains 117000 synsets connected through their semantic sense. WordNet defines several relations such as synonymy, hypernymy, hyponymy, antonymy, meronymy, and troponymy. We will be using the synonymy relations between words in our implementation.

2.3 Grammatical Knowledge Patterns

As mentioned in Section 2.2, the typed dependencies generated by the Stanford Parser are used to identify the semantic relationships between words. These dependencies are used to extract relevant information from a complex sentence into intermediate data structures

Tag	Description
CC	conjunction, coordinating
CD	cardinal number
DT	determiner
EX	existential there
FW	foreign word
IN	conjunction, subordinating or preposition
JJ	adjective
JJR	adjective, comparative
JJS	adjective, superlative
LS	list item marker
MD	verb, modal auxillary
NN	noun, singular or mass
NNS	noun, plural
NNP	noun, proper singular
NNPS	noun, proper plural
PDT	predeterminer
POS	possessive ending
PRP	pronoun, personal
PRP$	pronoun, possessive
RB	adverb
RBR	adverb, comparative
RBS	adverb, superlative
RP	adverb, particle
SYM	symbol
TO	infinitival to
UH	interjection
VB	verb, base form
VBZ	verb, 3rd person singular present
VBP	verb, non-3rd person singular present
VBD	verb, past tense
VBN	verb, past participle
VBG	verb, gerund or present participle
WDT	*wh*-determiner
WP	*wh*-pronoun, personal
WP$	*wh*-pronoun, possessive
WRB	*wh*-adverb

Figure 2.1: Penn Treebank description of the POS Tags [31]

Tag	Description
NP	noun phrase
PP	prepositional phrase
VP	verb phrase
ADVP	adverb phrase

Figure 2.2: Penn Treebank description of Phrase Tags [31]

called *frames* using Grammatical Knowledge Patterns (GKP) based on [35]. Grammatical Knowledge Patterns combine groups of word patterns together based on their parts of speech. We will use the linguistic parameters selected by [35] for identifying the GKPs. These parameters are *"Structure of sentence," "Special Parts of speech (e.g. Preposition, Markers, Conjunctions, etc.)," "Precondition Keywords (e.g. after, before, if, etc.)"*. The description of the identified GKPs is as follows:

1. Structure of sentence: Active or Passive

   - Active voice is detected if the sentence follows the form:

Active Voice	**<subject> <main verb> <object>**

   - Passive voice is detected if the sentence follows the form:

Passive Voice	**<form of TO BE> <verb in PAST PRINCIPLE>**

   The dependency tags from the parser output are used to extract the patterns above.

2. Special Parts of speech (e.g. Preposition, Markers, Conjunctions, etc.)

   - Preposition

     A preposition precedes nouns or pronouns and expresses a relation to other words or phrases. The preposition object is the noun or pronoun introduced by the preposition. The pattern for identifying preposition follows the form:

Preposition	**<clause> <NOUN/PRONOUN/PHRASE><PREPOSITION> <PREPOSITION OBJECT> <clause>**

   - Markers

     Markers are linking words such as because, and, but, or which indicate the grammatical function of the linked words or phrases. Markers are identified using the following pattern:

Marker	**<clause> <MARKER_KEYWORD> <clause>**

   - Conjunction

     Conjunctions connect words, phrases, or sentences of similar syntactic structures. Coordinating conjunctions connect verbs or nouns. The corresponding patterns are in the following form:

Verb	**<clause> <verb_1> <CONJUNCTION> <verb_2> <clause>**
Noun	**<clause> <noun_1> <CONJUNCTION> <noun_2> <clause>**

3. Precondition Keywords (e.g. after, before, if, etc.)

   A precondition is an existing clause that must be fulfilled for the dependent clause to be valid. The patterns for preconditions are as follows:

Precondition	**<AFTER/ ON/ONCE/ HAVING> <Precondition clause> <Dependent clause>**
	<IF> <Precondition clause> <THEN> <Dependent clause>
	<HAVING> <verb in PAST PARTICIPLE> <Precondition clause> <Dependent clause>

The sentences formed by extracting relevant information are displayed as activities in the activity diagram and are used to create the activity graph. The activity graph is traversed using the DFS algorithm to generate functional test cases.

2.4 Generation of Activity Diagrams and Test Cases

The complete implementation of the proposed systems contains three different processes. These are (i) Generation of activity diagrams from Agile user stories. (ii) Generation of test cases from activity diagrams. (iii) Generation of test cases from Agile user stories. There has been a substantial amount of research done in the generation of activity diagrams from natural language requirements and generation of test cases from activity diagrams, which has influenced the design of our system as explained in Sections 2.2 and 2.3. Based on the previous work, it is observed that generation of test cases from Agile user stories using natural language processing is a relatively unexplored field, and this is the process that our implementation is targeting.

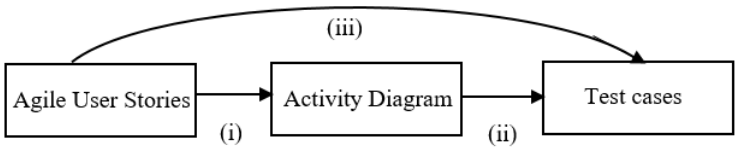


Figure 2.3: Different processes involved in test case generation

2.4.1 Generation of activity diagrams from natural language requirements

In Chapter 3, we use the concept of parsing the user requirements into intermediate frame structures using Grammatical Knowledge Patterns to extract relevant information from a sentence and create the activity diagram. This approach is based on Bhatia et al.'s [35] work which discusses the generation of frames using Grammatical Knowledge Patterns (GKP) from natural language requirements. The paper discusses identifying GKPs that can capture the essential parts of a sentence. The GKPs employed in the paper are "Structure of sentence," "Special Parts of Speech," and "Precondition keywords." The patterns were identified from analyzing 70 active voice sentences and 40 passive voice sentences. The sentences were lexically and syntactically analyzed using the Stanford Parser to determine the dependency tags and identify the frame keys which are the modifiers of the statement. The frame structures are populated based on the frame keys and dependency tags. The paper uses an older version of the Stanford Representation, and the frame keys utilized in this paper need to be modified for our implementation. The frame representations generated in this work were used in [23] to create activity and sequence diagrams. The input to the system are requirements, written in the form of paragraphs, not specified in the Agile user story format. The output of this work is an activity diagram depicting the flow of activities in the defined user story.

2.4.2 Generation of test cases from activity diagrams

There are several papers which discuss the generation of test cases from a UML diagram such as activity diagram, sequence diagram, and use case diagram [19–22]. We will be focusing on research that involves generation of test cases from an activity diagram. Within the scope of activity diagrams, there are many approaches for generating test cases such as Gray-Box method [36], sub-activity diagram [5], and Concurrent activities [37]. The implementation of this work has derived ideas from the papers which generate test cases from an activity diagram using the Depth First Search algorithm [7, 19, 37]. Another important point to be noted is that all of the research involving test case generation from an activity diagram inputs the activity diagram using a UML modeling tool such as IBM Rational Rose, Visual Paradigm for UML, Enterprise Architect, Magic Draw UML, ArgoUML, etc. In contrast, our work creates an activity diagram from user requirements.

Jena et al. [7] proposed a genetic algorithm for generating test cases from an activity diagram. The input to the system is the activity diagram for the ATM Withdrawal case study. An activity flow table is created using the activity diagram and then transformed into Activity Flow Graph (AFG). All the paths in the graph are traversed using the Depth First Search (DFS) algorithm and then printed as the test cases. The activity coverage criterion is used to evaluate the coverage of the test paths. A genetic algorithm is used to optimize the test

cases and eliminate redundant ones. The paper mentions constructing an activity diagram for a problem statement but does not delve into NLP. The paper only considers the ATM Cash Withdrawal System which is a popular example used to assess activity diagrams and their test coverage in other studies [7, 38]. The goal of the ATM cash withdrawal system is to dispense cash when the user inserts a valid card and PIN in the ATM. The system performs an authentication check and checks for availability of funds in the user's account. If sufficient funds are available, the system dispenses the cash, prints the receipt and completes the transaction.

Boghdady et al. [19] propose automated generation of test cases from an XML-based UML activity diagram. The XML schema is generated for the activity diagram uploaded using a UML modeling tool. The XML schema is used to create an Activity Dependency Table (ADT) which in turn creates the Activity Dependency Graph (ADG). Depth First Search (DFS) is applied to the ADG to obtain complete test paths. A hybrid coverage criterion consisting of branch coverage criterion, condition coverage criterion, the cyclomatic complexity coverage criterion and the full path coverage criterion is used to evaluate the generated test paths. The generated test cases are reduced by combining the test cases generated for loops.

Both these papers follow a similar method of applying DFS to the activity diagram to generate test paths and test cases. We will be using the same method of applying DFS to obtain test paths, but we will be using activity diagrams and frame structures created by our tool as inputs to the process.

2.4.3 Generation of test cases from Agile user stories

To the best of our knowledge, there is only one paper which discusses the idea of generating functional test cases from Agile user stories. Elghondakly et al. [39] proposed a requirement-based testing approach for automated test generation for Waterfall and Agile models. This proposed system would parse functional and non-functional requirements to generate test paths and test cases. The paper proposes the generation of test cases from Agile user stories but does not discuss any implementation aspects such as the techniques for parsing, or the format of the user stories that are parsed. This implementation does not follow a model based approach and does not evaluate the coverage of the test cases. The only parameter this paper uses to assess the test cases is the effort taken to create the test cases.

2.5 Research Contributions

As seen from the related work above, there are tools to generate activity diagrams from natural language or test cases from activity diagrams. But there is no existing tool to generate functional test cases from Agile user stories using natural language processing. In Chapter 3, we will see that our tool addresses the shortcomings of the other work by creating

frames structures using the Stanford Parser's latest Universal representation, eliminating the need for a UML modeling tool in test case generation and most importantly, providing a complete solution for generation of functional test cases from Agile user stories.

Chapter 3

Design & Implementation of Test Case Generation Tool

The high-level design idea is to automatically generate test cases from Agile user stories while staying as close as possible to the existing Agile software development process that is followed in the industry. The design of our system is guided by the procedures and terminology with which the Agile product owners, developers, and testers are familiar. However, the information provided by the user story is insufficient to create test cases using NLP, so we have introduced two new input parameters, *Test Scenario Description, and Dictionary*, to improve the test case generation process. The quality of the test cases generated by the tool is dependent on the test scenario description provided by the user. Guidelines detailed in Section 3.4.1 must be followed to ensure high-quality test cases which provide high test coverage. Sections 3.2 and 3.3 elaborate on the design constraints that we had to work with and our proposed system design to overcome the limitations.

In this chapter, we will discuss the implementation of the main contribution of this work, which is a tool for automatic test case generation. We will also examine the features of the system. In our research, we aim to generate test cases from natural language (NL) specification using the Graphical User Interface (GUI). User stories and test scenarios expressed in natural language act as the input to the system. Textual dependencies between the requirements are extracted from the tool using the Stanford Core Natural Language Processing (NLP) Suite. The dependencies are used to process relevant information (phrases, actions, activities, etc.) from the input and to structure them into frames using Grammatical Knowledge Patterns (GKP). Frames represent the test steps or activities and are used to create activity diagrams and activity graphs. The Depth-First Search (DFS) algorithm is used to iterate through the activity graph for test case generation. The system has been implemented using the Eclipse Integrated Development Environment (IDE).

Design of the system

3.1 Design Goals

The goal of the tool is to be able to generate functional test cases from Agile user stories within the boundaries of the Agile workflow. The *User Story* and *Acceptance criteria* are part of the existing Agile process and are currently used by the business owner to provide information about the features of the system to the developer. Our tool requires two pieces of additional information from the user which are the *Test Scenario Description* and the *Dictionary(Parameters, Test Steps)*. The test scenario description details the flow of information between the user and the system. The dictionary consists of commonly used keywords in the system and test steps which may break the system. Since this tool is not designed for a particular application or industry, the user can save the most commonly used test parameters relevant to their implementation in the dictionary. One of the reasons for extensive testing is to find instances where the code does not support the desired functionality. For example, in the case of a shopping website, numerical parameters such as "Quantity/Amount" are used very often. While testing numerical parameters, we ensure that the software does not crash for positive, negative, blank, or non-numerical values and these extreme cases are handled through an error handling mechanism. The dictionary needs to be set up just once and can be used for various features of the same application. Thus it will have little additional impact on the existing processes and on the time required to generate test cases automatically over the entire duration of a project. The purpose of the dictionary is twofold:

1. The dictionary function will help in incorporating extreme test steps which challenge the bounds of the code into the test cases.

2. In the case of a test scenario without any decision nodes, the flow of the process is linear and will generate only one test case, providing poor test coverage. The dictionary helps in increasing test coverage by branching out the test steps.

Functional test cases are the primary output of this tool. The intermediate output in the form of an activity diagram can be generated in a short time to represent the working of the system. The tool aims to reduce the time and effort spent in writing the test cases manually while improving the test coverage of the requirements.

3.2 Design Constraints

One of the main design constraints is that we want the tool to fit into the existing Agile workflow as much as possible. We have retained some of the Agile terminologies such as user story and acceptance criteria. As described in Section 2.1.3, the user story defines

the requirements of the design, and the acceptance criteria verify the completion of the development. However, the data gathered from the user story is not sufficient to obtain any meaningful NLP information. The user story is a one-line sentence in the As a/ I want/ So that format. This sentence does not provide a lot of information for natural language processing. We cannot obtain activities, user and system interactions using just this sentence. Hence, we ask the user to provide two additional inputs, i.e., the test scenario description and the dictionary. To reduce repetition, we have set up a database which stores and retrieves the relevant test scenario for a particular user story. The tool is highly dependent on the quality of the test scenario description and the dictionary parameters. While the system inputs free-form text, the users must follow the guidelines mentioned in Section 3.4.1 to obtain the correct output from the parser.

Another design constraint concerning the decision node in an activity diagram is that for an activity following a conditional statement, we ask the user to append the letter "Y" or "N" to the activity. After the conditional statement, the system has no way of knowing if the subsequent activities are to be executed for the THEN part or the ELSE part of the conditional statement.

The last design constraint related to the dictionary is that since we are not performing a context check on the test steps stored in the dictionary, we cannot modify the generation of the test paths. If a test case must terminate after a negative boundary condition, the logic must be input through the test scenario description. The system does not have the capability of recognizing the meaning of the test steps.

3.3 System Design

The constraints described in Section 3.2 have helped shape the design of our system and tool. The user provides four inputs related to the software feature being developed: User Story, Acceptance Criteria, Test Scenario Description, and Dictionary(Parameters, Test steps).

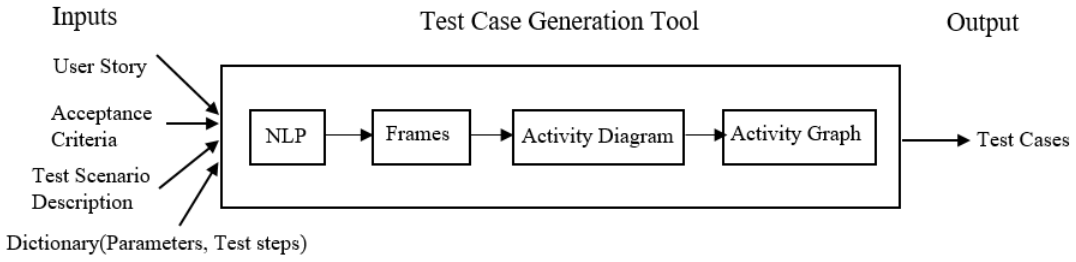


Figure 3.1: High-Level System Design

The system accepts these inputs and processes them using NLP techniques. The processed data is then stored in Frames to help in constructing activities. The activities are used to

create the Activity Diagram and Activity Graph. The activity graph will fetch related test steps from the dictionary. The DFS algorithm is used to traverse the activity graph and generate the final output which is the test cases.

Implementation Details

The Eclipse Integrated Development Environment has been used to develop this tool. Eclipse is an open source platform-independent development environment. Eclipse also provides the Abstract Window Toolkit(AWT), Swing, Standard Widget Toolkit(SWT) and Graphical Editing Framework(GEF) for developing graphical user interfaces in Java. We have used AWT, Swing, and SWT to implement graphical representations in this tool.

3.4 Graphical User Interface

The Graphical User Interface (GUI) facilitates the interaction between the user and the system. The GUI (Figure 3.2) was developed to assist testers in deriving functional test cases from the Agile user story. The GUI is created using Java Swing. Swing is a lightweight, easy-to-use GUI toolkit for Java. To understand the features of the tool, we will consider the example of an online shopping website where we browse through items, select and add them to our shopping cart.

3.4.1 Features of the GUI

The GUI has been developed keeping into consideration the process, terms and functionality already used in Agile Software Development and Agile Testing. The GUI consists of the following main components:

1. User Story

   Behavior Driven Development (BDD) provides user stories as a means of communicating the business outcome between the customer and developer. The business outcome is the expected behavior of the system represented as features. User stories realized the features by describing the interaction between the system and the user. The user story must clarify the role of the user, the feature to be implemented and the benefit derived from the feature. The pre-defined template for user stories provided by BDD is As a [**role**], I want [**functionality**], so that [**business value**]. The user story may be used in different contexts. Each context and outcome of the user story are called acceptance criteria in BDD [4].

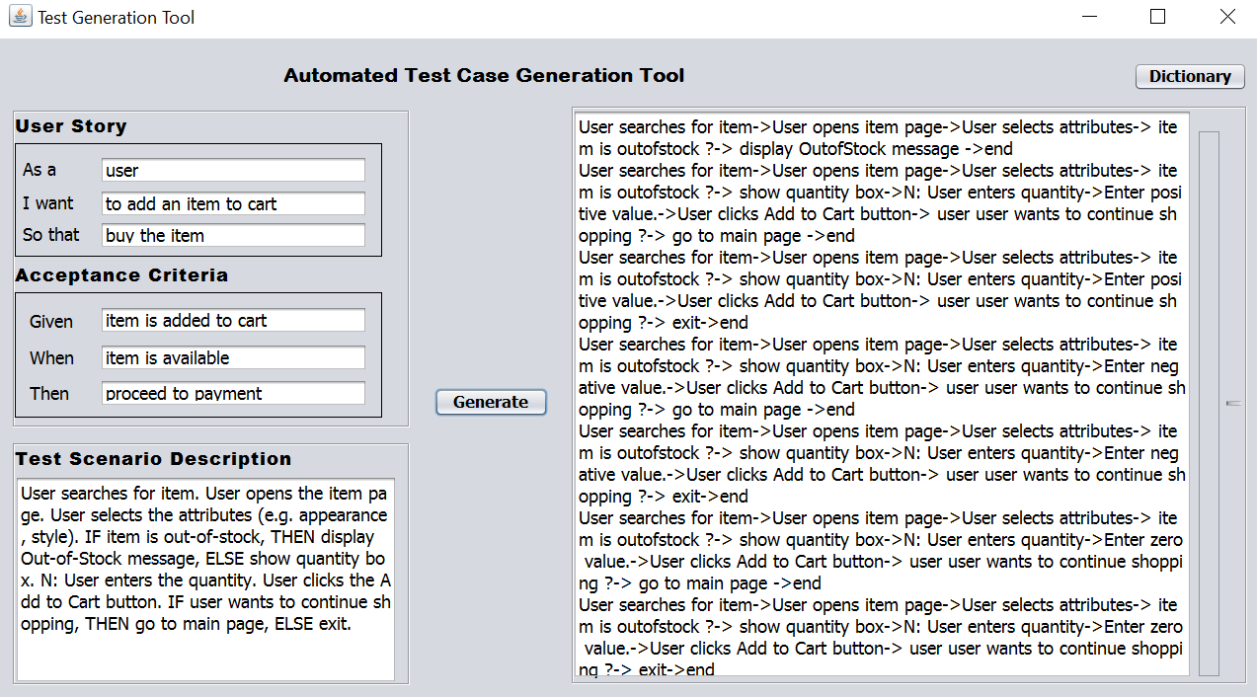


Figure 3.2: Automatic test generation tool

In the case of the online shopping cart example, our user story is *"As a user, I want to add an item to the shopping cart, so that I can buy the item."* In our tool, the role and the business value do not play a significant role in generating the test cases. The functionality part of the user story is parsed using the POS Tagger to retrieve its nouns and verbs which are matched with the saved test scenario descriptions. If a match is not found, the lemma and synonyms of the search words are used to compare to the existing test scenario.

2. Acceptance Criteria

   Acceptance criteria are the conditions used to determine if the product has met all the requirements specified by the customer. The criteria are a set of statements, having a pass/fail outcome, represented in a Given/When/Then format which defines the testing conditions for the related user story. There are several advantages to using acceptance criteria in test case generation: they showcase the functionality of the system from the user's perspective, resolve ambiguity in requirements and help in designing the tests which mark the completion of the user story.

   The acceptance criteria for the online shopping example can be *"Given that user can add an item to the shopping cart when the item is available, then proceed to payment."* The acceptance criteria are not used in this implementation. It is present in the design for the sake of completeness and to help in visually verifying if the generated test cases achieve the desired functionality.

3. Dictionary

   As discussed in Section 3.2, the dictionary stores keywords commonly used in user stories and their associated test steps. The dictionary helps in covering a wider range of test cases with the help of corner cases, edge cases and boundary cases which test the boundary conditions of the software. For example, for the keyword "Quantity," the tester would want to check the bounds of the parameter by using zero, positive and negative values. The dictionary helps provide a broad range of test cases for different parameters. Since the user stories that will be employed in the tool are not limited to a particular industry, the dictionary can be customized based on the functionality of the system and the industry. There are pre-existing keywords added to the dictionary. The user can add or delete keywords and test steps from the dictionary. The tool uses the keyword to extract the associated test steps and incorporates them in the test case generation.

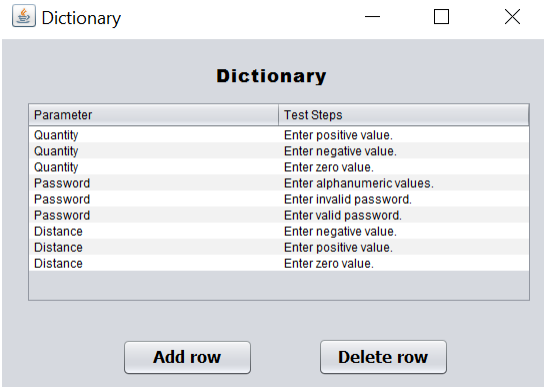


Figure 3.3: Dictionary

4. Test cases

   As the complexity of the software increases, it is challenging to test the system manually. Manual testing is prone to errors and is time-consuming, this raises the cost of the testing process [40]. Test cases are generated manually in the industry. The quality of the generated test cases depends on the knowledge transfer between the developer and tester, and on the tester's familiarity with the system. As described in Chapter 2, test case generation is one of the most time-consuming and tedious components of testing [8]. A test case may consist of test data, test steps, expected and actual outcome for a particular requirement. A test step describes the execution steps to verify the working of the system. A failed test step indicates that the developed system differs from the requirements. This tool generates test cases automatically from the test scenario description. Each test case consists of the test steps required to verify the behavior of the system and the compliance to the requirements.

5. Test Scenario Description

A scenario describes the sequence of interactions between the user and the system to achieve a specific goal. Scenarios are generally documented in textual form and have emerged as a popular method for requirements elicitation. The narrative nature of scenarios helps the user cover all aspects of the functioning of the system. The scenarios must consider the working of the system in normal cases and when the intended goal is not achieved. This system is supplemented by a Microsoft SQL Server 2016 database. The purpose of the database is to store the test scenario descriptions entered by the user for reuse at a later time so that the user does not have to rewrite the scenario repeatedly. Before the user enters a test scenario, the system analyses the user story and scans the database to check if a relevant test scenario exists in the system. If yes, the appropriate test scenario is retrieved from the database, else, the user enters a new test scenario. If search string matches an entry (or multiple entries) in the database, the results are retrieved and displayed to the user. The user can select the most appropriate test scenario description. On selecting a retrieved scenario, it is populated in the test scenario description box.

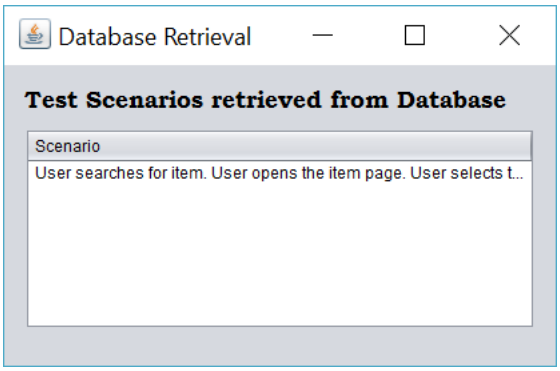


Figure 3.4: Retrieval from Database

Guidelines for writing the scenario

(a) The scenario must describe all the steps in the interactions between the user and the system relevant to the feature being developed.

(b) The interactions must be listed in a sequential manner, in the order, they are performed to achieve the functionality being developed.

(c) Each interaction must be listed as a new sentence.

(d) The steps must be written in simple sentences to reduce ambiguity.

(e) Avoid the use of pronouns since most parsers have issues resolving pronouns.

(f) Ensure consistent language between the user story, dictionary and test scenario description.

(g) The application being developed is referred to as the "system."

(h) If a conditional statement exists, where the system has multiple paths of action available, write the conditional statement in the following format:
IF {condition}, THEN {action 1}, ELSE {action 2}

(i) For interactions following the conditional statement, append the activity with "Y:" or "N:" to indicate if they are the logical successors to the THEN part or the ELSE part of the conditional statement.

An example of a scenario for adding an item from an online shopping portal to the shopping cart.

"User searches for item. User opens the item page. User selects the attributes (e.g. appearance, style). IF item is out-of-stock, THEN display out-of-stock message, ELSE show quantity box. N: User enters the quantity. User clicks the Add to Cart button. IF user wants to continue shopping, THEN go to main page, ELSE exit."

6. Multiple User Stories per Scenario Description

A test scenario description describes the process flow of the entire system. Multiple user stories may be associated with one scenario description. A test scenario description can be reused for multiple user stories. Consider the example where a scenario S1 is relevant for user stories U1, U2, and U3. S1 describes all the activities related to the feature being developed.

Let us assume that there are 10 activities in the system.

S1 = {A1, A2, A3, A4, A5, A6, A7, A8, A9, A10}

The user stories U1, U2, and U3, are linked to different sets of activities from S1. These activities are sequential and require the user to proceed in order. User stories related to a particular feature also follow a similar pattern.

U1 = {A1, A2, A3, A4}

U2 = {A1, A2, A3, A4, A5, A6, A7}

U3 = {A1, A2, A3, A4, A5, A6, A7, A8, A9, A10}

We would like to generate test cases for these user stories using the same scenario description since they are a part of the same feature of the system. The questions arise: how will the system understand which activities in the test scenario description correspond to which user story and how to avoid generating the same set of test cases for different user stories belonging to the same test scenario. To resolve this issue, we match the text in the user story with the test scenario description. When a match is found, the system will use parts of the test scenario description until the matching activity.

In the case of the test scenario description example explained above, there can be several user stories associated with it. For example,

"As a user, I want to select the size of an item."

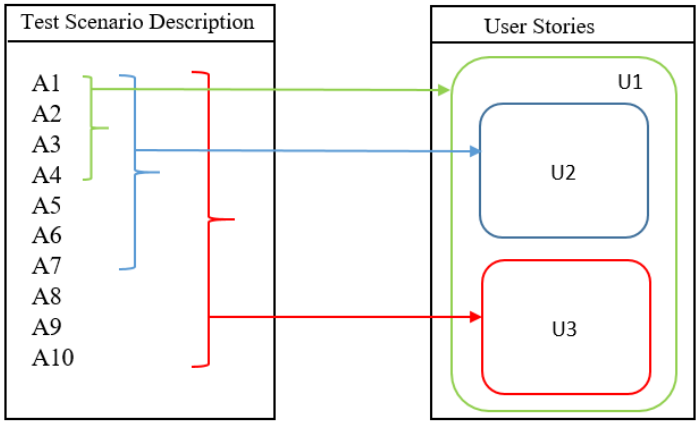


Figure 3.5: Mapping of Test Scenario Description to User Stories

"As a user, I want to enter the quantity of an item if the item is available."

"As a user, I want to add the item to the shopping cart."

Each of these user stories should generate different, relevant test cases from the same scenario description. Three methods are used for this purpose:

(a) Basic String Match

This is the simplest method of searching for the user story or parts of the user story in the test scenario description. The system looks for an exact match between the words of the search string and the scenario. The limitation of this method is that if there are multiple instances of the search string in the scenario description, the method has no way of identifying the right match.

(b) Lemmatization

Morphological analysis is performed on the words in the search string to extract the base of the word known as the lemma. The lemma is identified by removing the inflectional endings of nouns and verbs. Using this method, we can identify words related to or similar to the search phrase. For example, lexemes "go", "goes", "going", "went", "gone" have the lemma "go". The lemma of the search string is used to identify and demarcate the activity in the test scenario.

(c) Synonyms

Stanford Core NLP does not have the ability to produce synonyms of a word. Hence, we extract synonyms of a word using the lexical database called WordNet [33, 34] created by Princeton University. MIT's JWI (the MIT Java Wordnet Interface) [41] is a Java library used to interface WordNet and the Stanford Core NLP. Synonyms of the search string are used to identify and demarcate the activity in the test scenario.

These three methods together help us in the identification of the activities relevant to a user story and generating suitable test cases.

7. Error Handling messages

   There are several error handling mechanisms included in GUI to help the user navigate the tool. The purpose of the error handling messages is to let the user explore the features while understanding the limitations of the system.

   (a) The user can add or delete parameters and test steps from the dictionary. An error message is displayed if the user attempts to remove a row without selecting the row to be deleted.

   (b) The test scenario description is used to extract the textual dependencies between the words in the sentences. These textual relations are used to generate the functional test cases. An error message will be displayed if the user does not enter the test scenario description.

   (c) The user story is the key component of the tool. An error message is displayed if the user story or parts of the user story are not entered.

   (d) If the user provides an updated version of test scenario description, the system will overwrite the corresponding data entry in the database. The system informs the user that any changes to the scenario will update the existing one. The system also provides the user an opportunity to cancel the changes.

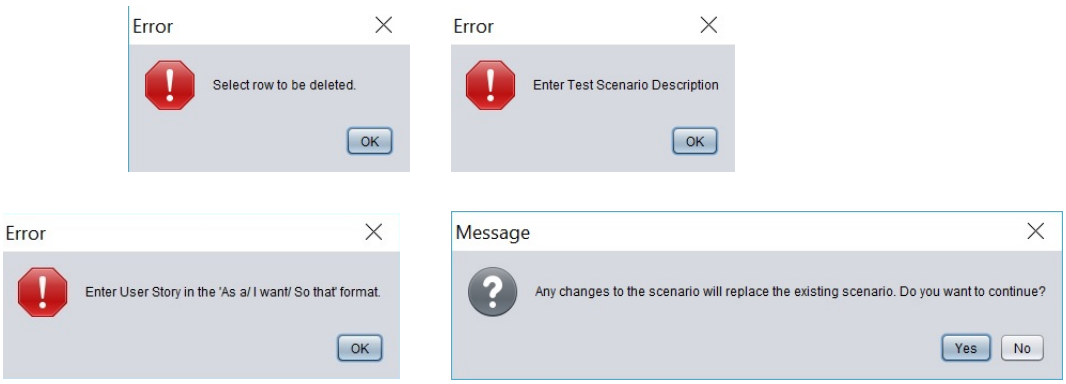


Figure 3.6: Error Handling messages

3.4.2 Using the Test Case Generation Tool

The user story, acceptance criteria, and test scenario description should be entered in their respective text areas. The user should define the application-specific corner cases using the Dictionary feature. Clicking the Generate button will trigger a search between the user

story and test scenario description. The user story, test scenario description, and dictionary parameters will be processed using the Stanford Parser, as explained in Section 2.2 to display the generated test cases.

3.5 Stanford Parser

NLP techniques described in Chapter 2, are used to parse the test scenario description. The first task in Agile user story to test case generation is the understanding of the test scenario description for the creation of a UML Activity Diagram representation that can be mapped to Activity Graphs and finally test cases. The scenario is used as the input to the NLP Parser instead of the user story since we can extract more information from the scenario.

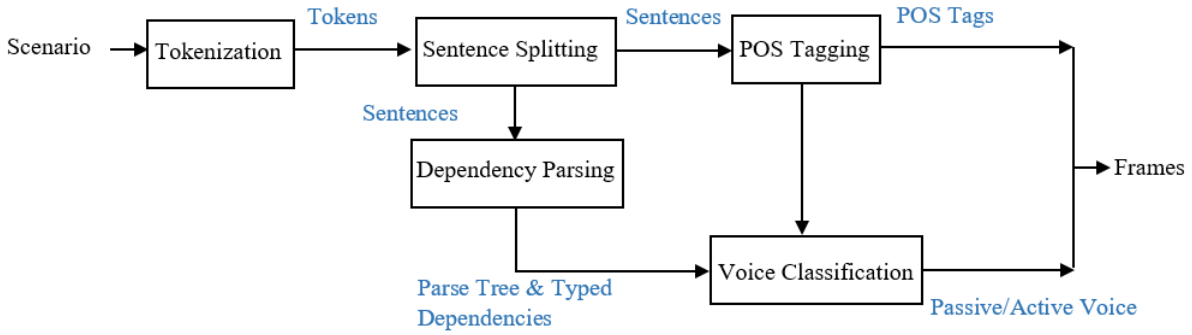


Figure 3.7: Natural Language Processing Block

This section briefly explains the mapping of a scenario expressed in English to an intermediate frame structure using the steps shown in Figure 3.7. The approach works in two phases: Lexical analysis and Syntactic analysis.

3.5.1 Lexical Analysis

In the preprocessing phase, lexical analysis of the input text containing the user story is performed using techniques such as tokenization, sentence splitting, and Part-Of-Speech tagging. The Stanford Parser and Stanford POS Tagger are employed for the preprocessing.

1. Tokenization

   In the first step, the test scenario description is provided as an input to the tokenizer, in the form of a paragraph. During the tokenization process, tokens, such as words, numbers, and punctuation, separated by spaces are identified in the input using the PTBTokenizer.

2. Sentence Splitting

   The sentence splitter reads the tokenized words as input and splits the paragraph into individual sentences. The boundaries of the sentence are identified, the text is divided into sentences, non-token words are eliminated, and each sentence is stored separately. Each sentence is terminated when a sentence ending character (., ! or ?) is found. The sentence splitting follows the tokenization so that each sentence of the paragraph in the form of tokenized words can be passed as the input to the POS Tagger and Dependency Parser.

3. Part-Of-Speech (POS) Tagger

   A POS tagger assigns parts of speech such as noun, verb, adjective, etc. to each word in the sentence using maximum entropy POS tagging [42]. The POS tagger uses the Penn Treebank tag set. The primary use of the POS tagger in this implementation is to help in the voice classification of each sentence. To illustrate POS tagging, we use the sentence "This is a simple sample sentence." The output of the POS tagger is shown in Figure 3.8.

English:	This is a simple sample sentence.
POS Tags:	[This_DT] [is_VBZ] [a_DT] [simple_JJ] [sample_NN] [sentence_NN].

Figure 3.8: Output of the POS Tagger

3.5.2 Syntactic Analysis

Syntactic analysis is performed using the Stanford Parser to parse the lexically analyzed text. The tokenized words, split into sentences is provided as the input to the dependency parser.

1. Dependency Parsing

   The Stanford parser analyses and extracts the grammatical structure of the sentence which indicates how the words are related to each other. It identifies the subject, object, verb phrase between the subject and the object, and conditional statements. The Stanford Parser uses a probabilistic natural language parser. The output of the parser (Figure 3.9 and 3.10) are the grammatical relations, also known as Stanford Dependencies and a phrase structure tree.

2. Voice Classification

   The output of the POS tagger and Dependency Tags are used to classify the sentence into active and passive voice categories. A sentence follows the active voice pattern if the subject of the sentence performs the action expressed by the verb. In a sentence

```
(ROOT
  (S
    (NP (DT This))
    (VP (VBZ is)
      (NP (DT a) (JJ simple) (NN sample) (NN sentence)))))
```

Figure 3.9: Output of the Dependency Parser - Phrase Structure Tree

```
nsubj(sentence-6, This-1)
cop(sentence-6, is-2)
det(sentence-6, a-3)
amod(sentence-6, simple-4)
compound(sentence-6, sample-5)
root(ROOT-0, sentence-6)
```

Figure 3.10: Output of the Dependency Parser - Typed Dependencies

using the passive voice, the subject is acted upon by the verb. Active voice can be identified by the presence of the nominal subject (nsubj) typed dependency in the sentence and passive voice by the presence of passive nominal subject (nsubjpass) typed dependency. Passive voice in a sentence can be identified by the use of a past principle tense with verbs and the use of the preposition "by" with the object [43]. The GKPs for recognizing the voice of the sentence are mentioned in Section 2.3.

On identifying the voice of the sentence, the POS tags and typed dependencies extracted above are used to search for prepositions, conjunctions, markers and pre-conditions in the sentence. These parts of speech are stored in temporary data holders called frames.

3.6 Frame Structure Representation

Frames are used as intermediary structured representations while transforming user requirements into behavioral UML models. Frame creation consists of two parts: identifying Grammatical Knowledge Patterns (GKP) in the test scenario description (refer Section 2.3) and using GKPs for populating the frames. Lexical analysis of the scenario description is performed to determine the GKPs. In this work, we fill the frames using the GKP Identification method proposed by Bhatia et al. [35]. All the grammatical patterns and frame structures used in this implementation have been taken from [35].

3.6.1 Frame structure

The test scenario description is analyzed using the POS Tagger and the Stanford Parser to extract the POS and dependency tags. POS tags help in the voice classification of the

sentence and the identification of preconditions and parts of speech. Sentences are classified as simple sentences and complex sentences. Simple sentences contain GKPs for active or passive voice. Complex sentences contain a simple sentence and can have one or more parts of speech such as prepositions, conjunctions or markers. Once the linguistic parameters present in the sentence are identified, the GKPs are used to generate frame structures for each sentence in the scenario. Frames consist of frame keys and dependency tags. Frame keys indicate the components of the sentence such as actor, the modifier of the actor, object, etc. Corresponding dependency tags are used to extract the values for the frame keys.

Table 3.1: Frame Structure for Active Voice.

Frame Key	**Dependency Tags**
Actor	NSUBJ/COMPOUND/ADVMOD(-,actor)
Modifier of Actor	AMOD(actor,?)
Action	ROOT/ACL:RELCL
Object	DOBJ(action, object)
Object Modifier	AMOD/ADVMOD/COMPOUND (obj , modifier)

Table 3.2: Frame Structure for Passive Voice.

Frame Key	**Dependency Tags**
Actor	AGENT(-,actor)
Modifier of Actor	AMOD(actor,?)
Action	ACL:RELCL
Object	NSUBJPASS
Object Modifier	DOBJ/XCOMP(action, object)

Table 3.3: Frame Structure for Preposition.

Frame Key	**Dependency Tags**
Preposition	CASE(-, ?)
Preposition Object	CASE(?, Preposition)
Preposition Governing Object	NMOD(?, Obj)
Modifiers	COMPOUND(Obj, ?)

Tables 3.1 and 3.2 define the frame structure for active and passive voice patterns. Tables 3.3 - 3.6 describe the frame keys and dependency tags for different parts of speech. The

frame keys are based on [35]. Since the publication of Bhatia et al. [35], Stanford parser has updated its representation to Universal dependencies. The dependency tags used in [35] do not give us the desired output. We have changed the dependency tags as required as shown in Tables 3.1 - 3.6 to extract relevant data based on our needs, which is also one of the limitations of the updated frames. These updated frames may not work for all possible sentences. The creation of frame structures which work for all complex English sentences is beyond the scope of this thesis. The quality of the sentences generated using frames is also dependent on the accuracy of the output of the Stanford POS Tagger and Stanford Parser.

Table 3.4: Frame Structure for Conjunction.

Frame Key	**Dependency Tags**
Conjunction	CC
Actor for Verb 1	NSUBJ/AGENT/COMPOUND(VERB1,?)
Actor for Verb 2	NSUBJ/AGENT/COMPOUND(VERB2,?)
Object for Verb 1	DOBJ/NSUBJPASS(VERB1,?)
Object for Verb 2	DOBJ/NSUBJPASS(VERB2,?)

Table 3.5: Frame Structure for Precondition.

Frame Key	**Dependency Tags**
Precondition	MARK(-, ?)
Precondition action	MARK(?, precondition)
Object of precondition	DOBJ(action,?)
Precondition on action	ACL:RELCL(obj, ?)

Table 3.6: Frame Structure for Marker.

Frame Key	**Dependency Tags**
Marker	ADVMOD(-, ?)
Marker Action	ADVMOD(?, Marker)
Marker Object	NSUBJPASS(Action,?)

All the relevant frame structures are concatenated to create a reconstructed sentence with necessary information. The data in the frames is used to generate the activity diagram. The nouns and verbs detected in each frame are used to search the dictionary for matching

values. If no match is found, the lemma and synonyms of the search words are used to compare with the dictionary parameters. The matched values are retrieved and added as nodes in the activity graph. The DFS algorithm traverses all the nodes including the nodes added from the dictionary.

3.7 Test Case Generation

The information extracted from the test scenario description and stored in the frame structure is used to generate the test cases. The restructured sentences constructed from the frames is used to create the Activity Diagram. The Activity Diagram is then used to create the Activity Graph, which is the graphical representation of the Activity Diagram. The paths in the Activity Graph are traversed using the Depth First Search algorithm to generate the test cases.

3.7.1 Activity Diagram

Activity Diagram depicts the flow of interaction between objects in UML using activities. Actors, objects, prepositions, and pre-conditions are used to extract the action phrase using frames. These phrases describe the sequence of interactions between the user and system. The layout and rendering package called Draw2D provided by Eclipse Graphical Editing Framework (GEF) is used to generate the Activity Diagram.

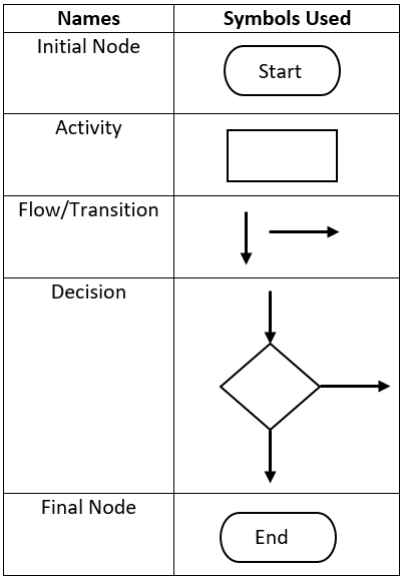


Figure 3.11: Constructs used in an Activity Diagram

The activity diagram consists of the initial node, activities, transitions and the final node. Parallel flow of activities (concurrency) and looping back to previous activities is currently not supported by the tool. All the activities are executed sequentially. The initial node indicates the start of the activity and the final node shows the termination of the activity. The initial and final node are denoted with rounded rectangles. The flow of the process or the transition between activities is shown using arrows. The activity node describes the process or behavior. It is represented using a rectangle. Conditional statements are depicted using the decision node represented as a diamond. The decision node has one incoming edge and multiple arrows labeled with the boolean expression that must be satisfied for the branch to be selected. The flow of the process has multiple paths to chose from, but it can traverse at most one of them. The flow must take one direction only. The boolean expressions must be mutually exclusive of one another so that only one of them can be true at a given point in time. If the data retrieved from the frame is a process statement, the activity node is used. The decision node is used for conditional statements in the IF/THEN/ELSE format. Figure 3.12 shows the activity diagram generated for the online shopping example using the user story, acceptance criteria, test scenario description and dictionary parameters mentioned in Section 3.4.1.

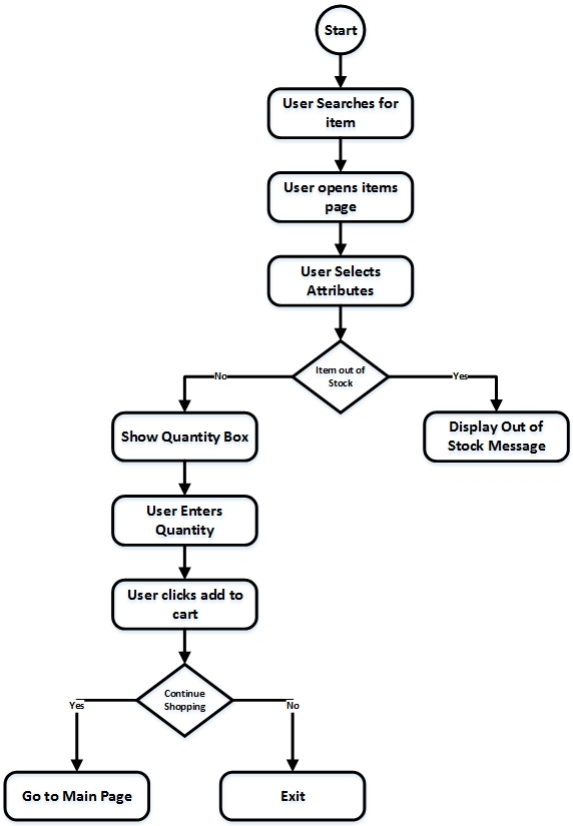


Figure 3.12: Activity Diagram for the Online Shopping example

3.7.2 Activity Graph

Graphs are a mathematically precise visual method of modeling objects and the relation between them. Activity Diagrams can be considered as a graph, with the activities represented as nodes and transitions as edges. Activity Diagrams are converted into activity graphs for the purpose of visualization and to implement the Depth First Search (DFS) algorithm. Figure 3.13 shows the activity graph created by the tool for the activity diagram in Section 3.7.1. As we can see below, the activity diagram integrates the test steps from the dictionary (Enter positive, negative and zero values) as nodes 7, 8 and 9.

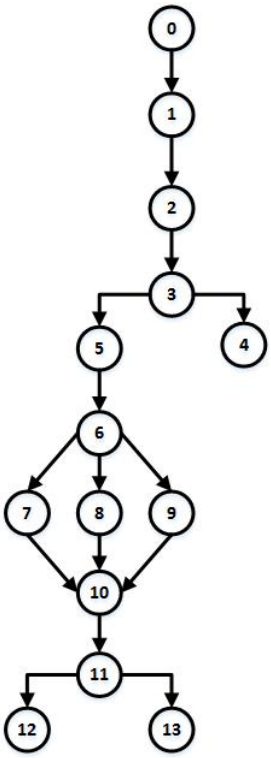


Figure 3.13: Activity Graph for the Online Shopping example

Each element in the graph is called a vertex, and the connection between the vertices are called edges. A path is a connection between two vertices using a sequence of edges. The activity graph is a variation of a directed graph. A directed graph is a graph where all the relationships between the vertices point in one direction. Since the activity diagram is a sequential flow diagram, all the paths in the activity graph will lead towards the next activity. Each node will have a single edge coming towards it but can have multiple edges emerging from it. The activity graph incorporates the Dictionary feature while generating the paths. The activity graph searches through the Dictionary to match the keywords stored in the dictionary with the words of the activity. If a match is found, the system retrieves the matched corner cases stored in the dictionary and adds them to the activity graph. DFS

algorithm is used to search all nodes, from the initial node to the final node, to calculate all the paths of an activity diagram.

3.7.3 Depth First Search Algorithm

Depth First Search (DFS) is a graph traversal algorithm that starts at an arbitrary root node and travels downward depth-wise till the last possible node before backtracking. The DFS algorithm considers the nodes to be unvisited (u) before they are traversed and visited (v) once they are traversed. The algorithm starts at a root node, moves to the next adjacent unvisited node. It marks the node as visited (u $->$ v). Then it proceeds recursively to the next node until it reaches the last node. At this point, the algorithm backtracks to previously visited nodes. Backtracking is moving forward until there are no unvisited nodes along the current path, then the algorithm traverses backward on the same path to find other unvisited nodes. The next path will be selected after all the nodes on the current path are visited. We must keep track of the visited nodes so that we do not visit the same node twice and we can trace our steps back. After the creation of all test paths, we generate and print the test cases from the paths. Figure 3.14 shows all the test paths and test cases generated from the activity graph by applying the DFS algorithm.

Test Paths and Test Cases for Online Shopping Example
0 → 1 → 2 → 3→ 4 User searches for item → User opens item page → User selects attributes → item is outofstock? → display OutofStock message → end
0 →1→ 2→ 3→ 5→ 6→ 7→ 10→ 11→ 12 User searches for item → User opens item page → User selects attributes → item is outofstock? → show quantity box → N: User enters quantity → Enter positive value → User clicks Add to Cart button → user wants to continue shopping? → go to main page → end
0→ 1→ 2→ 3→ 5→ 6 →7→ 10→ 11 →13 User searches for item → User opens item page → User selects attributes → item is outofstock? → show quantity box → N: User enters quantity → Enter positive value → User clicks Add to Cart button → user wants to continue shopping? → exit → end
0 →1→ 2→ 3→ 5→ 6→ 8→ 10 →11→ 12 User searches for item → User opens item page → User selects attributes → item is outofstock? → show quantity box → N: User enters quantity → Enter negative value → User clicks Add to Cart button → user wants to continue shopping? → go to main page → end
0→ 1 →2 →3 →5→ 6→ 8→ 10 →11→ 13 User searches for item → User opens item page → User selects attributes → item is outofstock? → show quantity box → N: User enters quantity → Enter negative value → User clicks Add to Cart button → user wants to continue shopping? → exit → end
0 →1→ 2→ 3→ 5→ 6→ 9→ 10→ 11→ 12 User searches for item → User opens item page → User selects attributes → item is outofstock? → show quantity box → N: User enters quantity → Enter zero value → User clicks Add to Cart button → user wants to continue shopping? → go to main page → end
0 →1→ 2→ 3→ 5→ 6→ 9→ 10 →11→ 13 User searches for item → User opens item page →User selects attributes → item is outofstock? → show quantity box → N: User enters quantity → Enter zero value → User clicks Add to Cart button → user wants to continue shopping? → exit → end

Figure 3.14: Test paths and test cases for the Online Shopping example

Chapter 4

Evaluation of Test Case Generation Tool

This chapter describes the methods used to evaluate the Test Case Generation Tool. Since our implementation creates test cases based on the user stories and activity diagram, it combines Requirement based testing and Model-based testing to generate our test cases. To the best of our knowledge, there are no known parameters in existing research to evaluate the models created from natural language, apart from visual inspection of the model. While there are several test coverage criteria available to validate the generated test cases, most of these techniques are suitable only for model-based testing. The model-based test coverage criteria are mainly used for testing loops, faults in loops, and synchronization loops. These techniques focus on evaluating the model and not the test cases. The only parameters we have come across to assess the quality of the test cases generated from natural language or activity diagram are the number of test cases created and effort required to create the test cases. If the generated test cases are executed and linked to defects in the Agile software workflow, parameters such as rework ratio, defect detection percentage, and test execution rates can be used to verify the quality of the test cases.

We have used a two-step approach to evaluate the tool. First, we conducted a preliminary analysis of the designed test cases using the test coverage criteria used in Model-Based testing. We will use these criteria for the evaluation of the ATM Cash Withdrawal System described in Section 2.4. The parameters for evaluating test cases are the effort required to generate test cases and test case productivity. Next, we conducted a feasibility study to assess the usefulness of the tool. The results from each of these methods will be presented and analyzed in Chapter 5, Results and Discussion.

4.1 Preliminary Analysis of Test Cases and Model

4.1.1 Test Coverage Criteria

The following adequacy criteria determine where the testing process terminates and how good a test coverage does the generated activity graph provide since the test cases are generated from the activity graph [44].

1. Activity Coverage

   The generated test cases must ensure that all the activity states in the diagram are covered sequentially, from the initial node to the final node, at least once. Activity Coverage is calculated using the following formula:

   *(Number of activity states covered)/(Number of activity states present) * 100%*

2. Path Coverage

   The generated test cases must ensure that all possible paths from the initial node to the final node are covered at least once in the activity graph. Path Coverage is calculated using the following formula:

   *(Number of paths covered)/(Number of paths present) * 100%*

3. Transition Coverage

   The generated test cases must ensure that all possible transitions/edges from the initial node to the final node are covered at least once in the activity graph. Transition Coverage is calculated using the following formula:

   *(Number of transitions covered)/(Number of transitions present) * 100%*

4. Predicate Coverage or Branch Coverage

   In the case of a decision node, the generated test cases must cover the true and false logic paths of the condition. Branch Coverage is calculated using the following formula:

   *(Number of branch states covered)/(Number of branch states present) * 100%*

4.1.2 Effort required to generate the test cases

The effort required to generate the test cases is the average time taken to generate the test cases [39].

Effort = Time required to generate test cases / total number of generated test cases.

4.1.3 Test Case Productivity

Test Case Productivity (TCP) is defined as the ratio of the number of test steps/test case generated to the effort (in hours) taken to generate these test steps [45].

Test Case Productivity = (Number of Test Steps Prepared) / (Effort spent for Test Case Preparation)

We will be using these parameters to verify the working of the tool prior to the user study. The same parameters will also be applied in the analysis of the data gathered through the user study.

4.2 Developing a User Study

This section outlines the proposed user study to evaluate the efficiency, accuracy, and performance of the Test Case Generation Tool. This user study is created to test the feasibility of generating test cases from Agile user stories using natural language processing techniques. The user study addresses the questions posed in Chapter 1, Introduction.

1. Can we generate test cases from user stories in an Agile software development workflow using the test case generation tool?
2. Does this tool save the tester's time and effort while improving the quality and coverage of the test cases?

4.2.1 Feasibility Study Design

Participant Selection

To assess the effectiveness of the tool we conducted a user study with ten working professionals having knowledge or experience in software testing and are familiar with Agile software development. All the participants had backgrounds in computer science or computer engineering. They are currently working in various software firms as software developers or software testers. Some of the participants have experience with Agile testing. These participants were selected as they are representative of testers in the software industry that may find this tool useful. Ten participants were used for this study. Appendix B includes the IRB approval letter authorizing the user study.

Experimental Setup

Since the users are all working professionals located at different locations, we used video calling to conduct the study. The user was required to download the following software: a video calling software to participate in the user study, Marvel prototyping software to interact with the prototype applications created for the user study and a remote access software to access and use the test case generation tool since setup of this tool on each participant's computer is not possible. Three prototype applications were created using the Marvel app for the purpose of the user study. Each prototype had four features, of which two were tested manually by the user, and two were tested automatically. These prototypes were based on wearable applications:

1. AstroNav is an application which helps astronauts on a planetary surface navigate their way to a destination. The application displays the user's current location in terms of latitude and longitude. If the location is incorrect, the user can recalibrate the device. The destination can be selected from a list of previously used destinations or can be dictated using voice recognition or typed manually. On clicking navigate, the shortest and fastest route options are displayed to the user along with the hazardous condition information on each route. The main features are get current location, set the desired destination, view shortest and fastest route options and get hazardous condition alert.

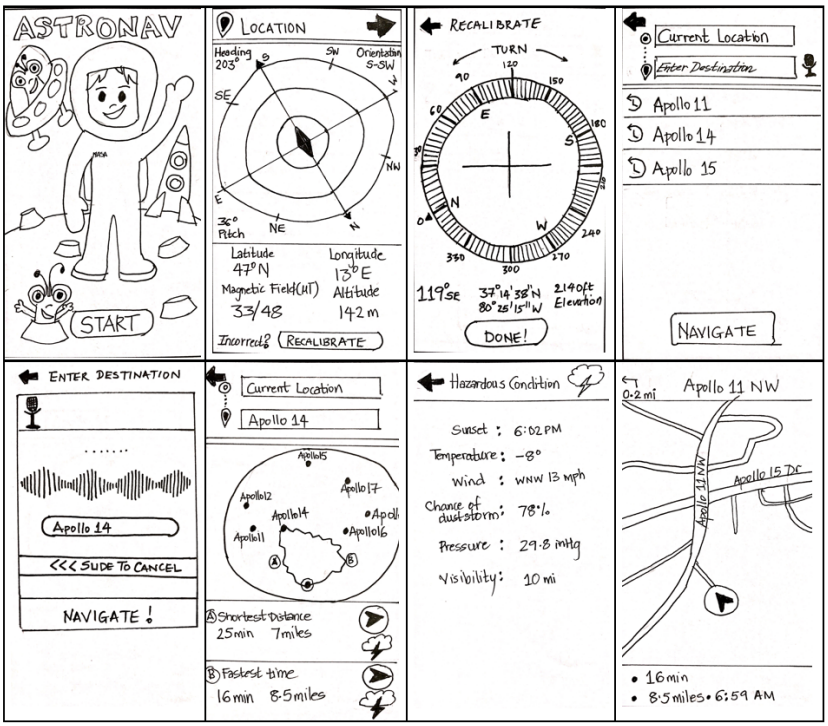


Figure 4.1: AstroNav Application

2. FitTrain is a fitness application in which trainers recommend exercises to the users. Trainers can log into their account and view their trainees or add a new trainee. Trainers recommend exercises to the trainee based on their height, weight and previous workout history. They can modify the schedule of the trainee by adding or removing workouts such as cardio and weights, or they can add specific exercises to the workout such as squats, push-ups, and laps. Users can log into their profile and select exercises for today from a list of exercises. They can also view their workout history and badges earned. The main features are modify exercise schedule, add a new trainee, view awards earned and set exercise goals.

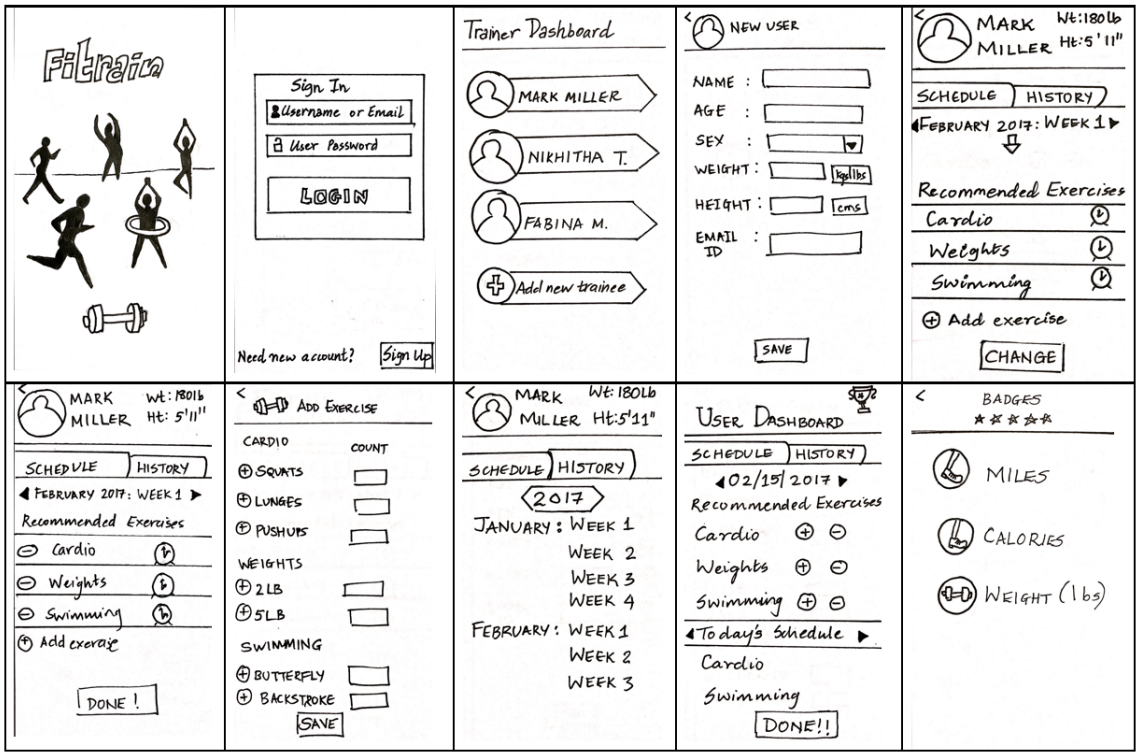


Figure 4.2: FitTrain Application

3. Mooditation is an application to monitor your stress levels using Heart Rate and to play mood-based music to calm you down. The application plays music based on the user's heart rate, stress levels, and detected mood. The user can view the heart rate data and stress level. The user can play the automatically detected music or select another song from the playlist. If the mapping of the mood to the song is incorrect, the user can change the mood associated with the song. The user can change the number of weeks and units by selecting the settings icon. The main features are logging into the profile, signing up for a new account, check historical stress level, listen to music based on stress level.

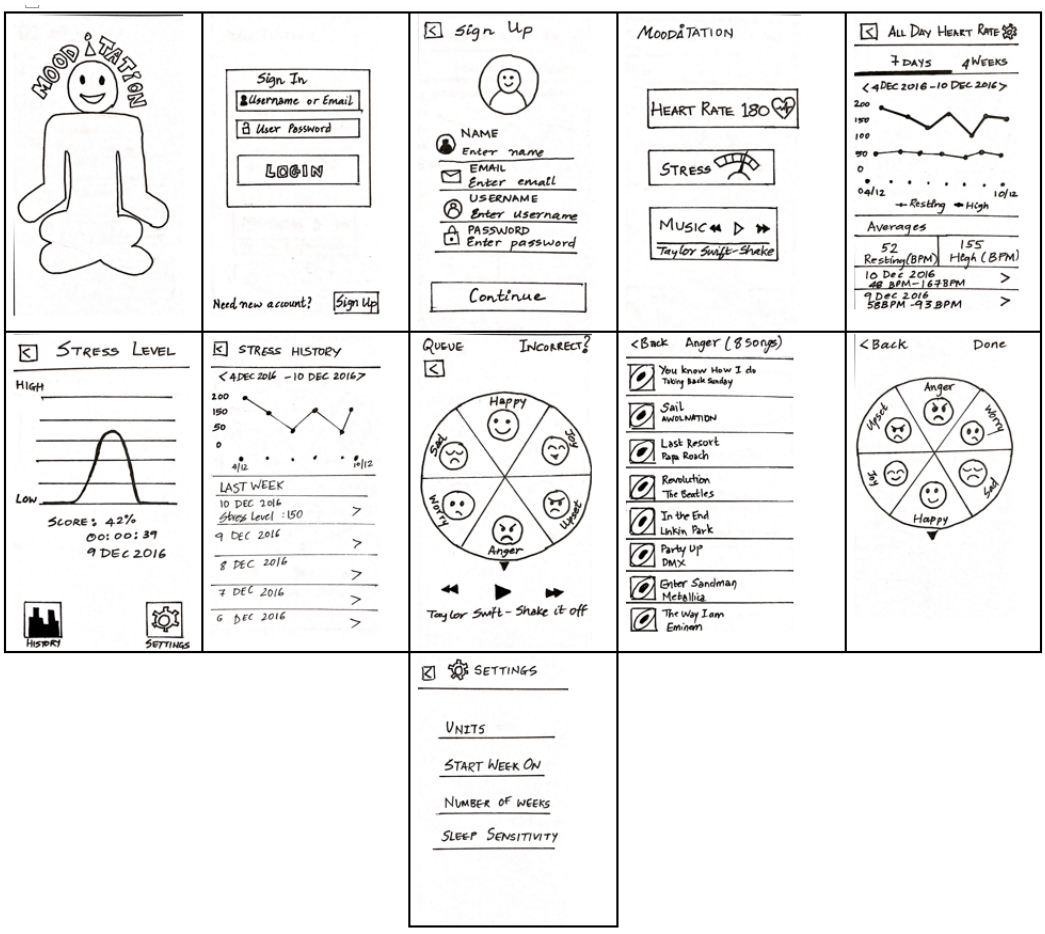


Figure 4.3: Mooditation Application

Since we had a total of 12 features, we randomly divided them into two sets, Set 1 and Set 2. Five participants tested Set 1 manually and Set 2 automatically. The other five participants tested Set 2 manually and Set 1 automatically. In this manner, each feature was tested both manually and automatically. For a fair comparison of manual and automatic testing, the user was given different features of approximately same complexity for both the processes. No feature was repeated for both manual and automatic testing for the same user.

Description of Metrics

The metrics measured in the feasibility study were time taken to generate test cases for a feature, number of test cases generated for each feature, number of test steps per test case, number of corner cases considered and the number of redundant test cases produced by the tool. These metrics are meant to gather insights about the accuracy, performance, and utility of the tool. The time taken to generate the test cases, both manually and automatically,

measures how much time the user takes to write all possible test cases for a given feature. In the case of the automatically generated test cases, it measures the time taken for the user to enter the parameters required to generate the test cases as well. The number of test cases measures the quantity of work that can be accomplished by the tester in a given amount of time. The number of test steps per test case and the number of corner cases indicate the quality and level of detail of the test case coverage. The number of redundant cases is measured to ensure that the tool is producing relevant and required test cases.

Assumptions and Limitations

The main assumption is that the user is using this tool within the scope of Agile software development. Agile software development breaks down large requirements into smaller features. These features called user stories have a relatively simple workflow. This tool currently does not support parallel or concurrent flows seen in more complex processes and hence uses only the simpler elements of the UML activity diagram.

There were multiple limitations in the feasibility study. First, the main limitation of the tool is that the quality of the test cases generated is largely dependent on the quality of the Test Scenario Description entered by the user. To overcome this limitation, we have developed a set of guidelines for the user on how to write the scenario.

Second, the tool will rely on the user's grasp of grammar and punctuation to be able to generate good quality results. The user must try to maintain the same language while specifying the user story and test scenario description. The tool does not understand context, so it will not be able to link the user story to the test scenario description if the manner of writing and words are different. The user must try to maintain consistency in words used in the test scenario description and dictionary as well. The tool will not be able to retrieve test steps from the dictionary if the boundary condition parameters or their synonyms are not specified in the test scenario description. Additionally, the sentences in test scenario description must be sequential. The system will not be able to understand the missing or incomplete logic in the test scenario description. To partially overcome this issue, we have used synonyms which will search for words having similar meaning, but the extent to which this solution will work is limited.

Third, the frames we have created may not generate grammatically correct sentences in all possible instances. Frame structures which work for all English sentences will require a detailed linguistic study which is beyond the scope of this thesis.

4.2.2 Study Description

Before the experiment, the user answers questions related to their experience with software testing and Agile methodologies. Next, the user accesses the prototype applications using

the Marvel App. The user was asked to interact with these applications and to get familiar with their features. The participant was given five minutes per application.

Next, the user was presented with two features per application (total six features) for which he/she has to write the test cases manually. While the user was writing the test cases, we record the data for parameters such as the time taken to generate test cases for a feature, number of test cases generated for each feature, number of test steps per test case, and number of corner cases considered.

Once the user was done with manual testing of the features, the participant was given a 10-minute training session on using the Test case generation tool. The training session explained the functionality of the tool, the main components which the user would have to enter i.e. user story, acceptance criteria, test scenario description, and dictionary. The guidelines for writing the test scenario description were explained to the user. A demonstration was conducted using a sample feature to get the user accustomed to the tool. After the training, the user was given two features per application (total six features) for which he/she had to generate the test cases using the tool. While the participant was using the tool, we recorded the metrics described earlier.

All the manually and automatically generated test cases and data entered into the tool were saved for post-survey analysis. After the experiment, the user was asked to answer questions related to the performance, accuracy, and utility of the tool. We were interested in knowing the user's preferred method in terms of time saved, better test coverage, less effort, and reusability. Since we asked the user for two additional inputs (test scenario description and dictionary) which are not part of the current Agile testing process, we asked them if the value addition provided by these features was worth the time spent in writing them. The participants were asked to submit their impressions about the accuracy, quality, and relevance of the generated test cases.

4.3 Data Analysis Techniques

4.3.1 Correlation

Correlation is a statistical method to estimate the strength of the relation between two variables. Correlation is measured in terms of correlation coefficient (R) which ranges from -1 to +1. If R is close to -1 or +1, it indicates a high correlation between the variables. If R is close to 0 there is no relationship between the variables. If the R value is positive, it indicates a direct correlation, as one variable increases the second variable will increase. A negative R value indicates inverse correlation, as one variable increases the second variable will decrease.

4.3.2 Analysis of Variance

Analysis of Variance (ANOVA) determines if there is a statistically significant effect of one parameter on another independent parameter. The important variables that are considered for the ANOVA test are the p-value and alpha. The alpha defines the level of confidence with which we can ascertain the results of the ANOVA test assuming a normal distribution. The ANOVA test depends on the value of p. If p-value < (1- alpha) the variable under consideration has a significant effect on the other variable. If p-value > (1- alpha) then the effect is considered insignificant. Alpha is 0.05 for this study, providing a 95% confidence level.

4.3.3 Hypothesis Testing

Hypothesis testing or t-testing is a method of statistical analysis to determine if a certain premise or hypothesis is true for a larger population. If the result of the t-test is true, it indicates that the results are statistically significant, and the results are not random. The process of hypothesis testing consists of the following steps:

1. The hypothesis is expressed as the null hypothesis and the alternative hypothesis. The null hypothesis (H0) assumes that the hypothesis is not true. The alternative hypothesis (H1) is the premise that we are trying to prove.

$$H0 = \mu1 - \mu2 < 0$$

$$H1 = \mu1 - \mu2 > 0$$

2. The level of statistical significance is the p-value. Set the p-value which is the probability that the test parameter is significant if the null hypothesis is true.

3. The p-value is compared to an acceptable significance value i.e. 0.05. If p <=0.05, that the observed effect is statistically significant, reject the null hypothesis, and accept the alternative hypothesis as valid with 95% confidence level. If p> 0.05 then we fail to reject the null hypothesis, but we do not have conclusive evidence to accept the null hypothesis.

Chapter 5

Results and Discussion

In this chapter, we will present the results of the preliminary analysis and the feasibility study described in Sections 4.1 and 4.2. The results of the preliminary analysis are indicative of whether the test cases provide good test coverage and if the tool can generate accurate and high-quality test cases. The preliminary analysis evaluates the performance of our tool for the ATM Cash Withdrawal example described in Section 2.4. The results of the feasibility study tell us if the generated test cases save time, effort and increase the test case productivity of testers using this tool for Agile testing.

5.1 Preliminary Analysis

We used the parameters described in Section 4.1 to evaluate the ATM cash withdrawal system [7, 38]. Oluwagbemi et. al. [38] compares and summarizes nine different test case generation approaches based on activity diagrams. All the studies generating test cases from Activity Diagrams are using tools such as IBM Rational Rose, UML 2.0, ArgoUML to create and input the activity diagram. Using UML tools allows the creation of complex diagrams that are not supported by our tool. As mentioned in the Section 3.7.1, our tool does not support loops and concurrent behavior. Our tool supports the essential elements such as Activity, Transition, and Decision node without concurrency and looping. Our tool creates the activity diagram from the test scenario description using natural language processing.

In the case of this example, the test scenario description we used is *"User inserts Card. User enters PIN. IF user authorized THEN Select Account Type ELSE Eject card. Y: Enter Amount. System checks balance. IF amount<balance THEN Debit Amount ELSE Show Error. N: Eject card. Y: System disperses cash. System prepares printer. System prints receipt. Eject Card"*. Three test steps were set up in the dictionary for the parameter "Quantity." They were "Enter positive value, Enter negative value and Enter zero value." Since the words quantity and amount are synonyms, the activity graph compared the test

scenario description to the dictionary and fetched the test steps. The test steps retrieved from the dictionary are not added to the activity diagram.

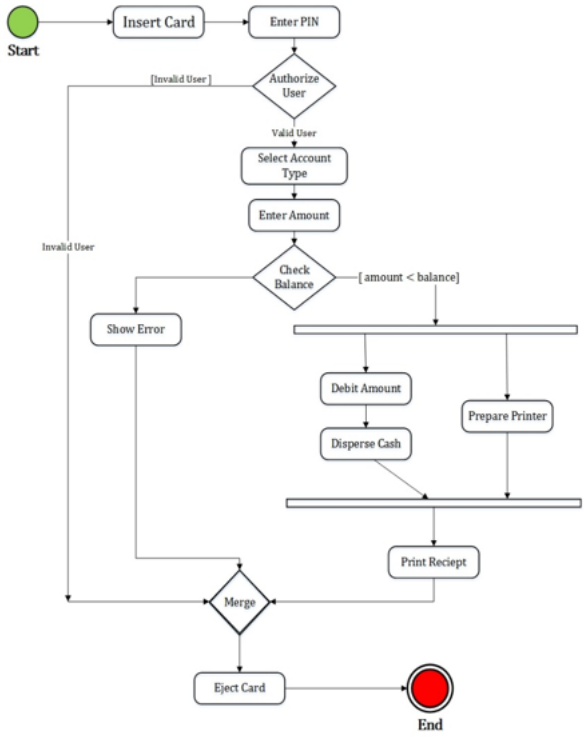


Figure 5.1: Activity Diagram for ATM cash withdrawal example [7, 38]

Figures 5.1 and 5.2 show the Activity Diagrams for ATM Case Withdrawal Case Study, based on [7, 38] and using our tool. Through the preliminary analysis, we want to compare the activity diagram generated by our tool to the activity diagram used by [7, 38]. Their activity diagram uses concurrent flows, fork and merge elements, which are not supported by our tool. As shown in figure 5.2, even without concurrent activities and looping, our activity diagram covers all the states covered by the original activity diagram. A detailed comparison of test coverage and test cases is presented in the next section.

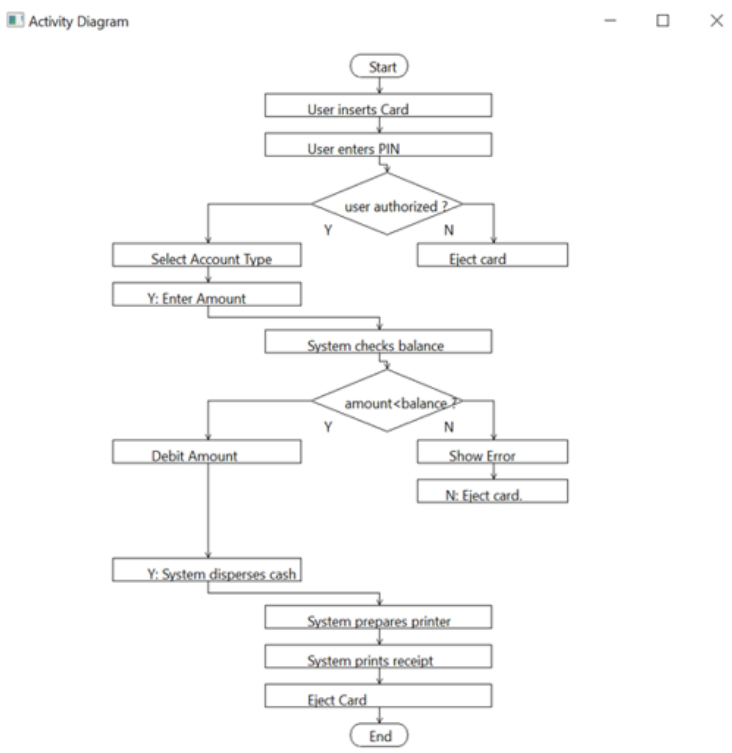


Figure 5.2: Activity Diagram for ATM cash withdrawal example using our tool

5.1.1 Analysis of Quantitative Data

We will be using the test coverage criteria discussed in Section 4.1 to evaluate the performance of our tool prior to the feasibility study. The test coverage criteria that we obtain for the ATM cash withdrawal example using our tool is as shown in Figure 5.3. The number of elements covered is counted based on the number of activities, branches, paths, conditions and transitions covered by our activity diagram. The number of elements present is obtained from the activity diagram from [7, 38].

Element	Number of Elements Present	Number of Elements Covered	Coverage
Activity	10	10	100%
Basic Paths	3	3	100%
Transitions	14	14	100%
Branches	2	2	100%
Conditions	4	4	100%

Figure 5.3: Test coverage criteria for ATM Cash Withdrawal case study

The test cases generated by the tool for this example are shown in figure 5.4. Figure 5.5 compares the number of corner cases and redundant cases generated by our tool and other implementations [7]. The number of corner cases is the number of boundary conditions set

up in the dictionary for a parameter. The number of redundant cases is the number of invalid test cases generated by the tool. Jena et. al. [7] uses a genetic algorithm to optimize their test cases and minimize redundancy.

Test Cases for ATM Cash Withdrawal
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter positive value → System checks balance → amount<balance? → Debit Amount → Y: System disperses cash → System prepares printer → System prints receipt → Eject Card → end
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter positive value → System checks balance → amount<balance? → Show Error → N: Eject card → end
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter negative value → System checks balance → amount<balance? → Debit Amount → Y: System disperses cash → System prepares printer → System prints receipt → Eject Card → end
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter negative value → System checks balance → amount<balance? → Show Error → N: Eject card → end
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter zero value → System checks balance → amount<balance? → Debit Amount → Y: System disperses cash → System prepares printer → System prints receipt → Eject Card → end
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter zero value → System checks balance → amount<balance? → Show Error → N: Eject card → end
User inserts Card → User enters PIN → user authorized? → Eject card → end

Figure 5.4: Test cases generated by the tool for ATM Cash Withdrawal case study

Parameter	**Test cases [6,32]**	**Test cases – our tool**
Corner cases	0	3
Redundant cases	0	0
Number of test cases	5	7

Figure 5.5: Comparison of Test cases for ATM Cash Withdrawal case study

We will use the metrics described in section 4.1 to evaluate the performance of our tool for the ATM case study. The time taken by us to enter the user story, acceptance criteria, test scenario description, dictionary parameters and generate the test cases was approximately 4 minutes (rounded off to the nearest integer). The number of test cases generated is 7, and the number of test steps in the test cases is 70. The effort is indicative of the average time required to generate each test case. Test Case Productivity (TCP) tells us the number of test steps we can generate per hour of effort. The effort is converted to hours. Using the formula described in Section 4.1, we calculate the effort and TCP:

Effort = Time required to generate test cases / total number of generated test cases
Effort = 34 seconds/test case

TCP = Number of Test Steps Prepared / Effort spent for Test Case Preparation
TCP = 1044 Test steps/hour.

5.1.2 Discussion of Results

Despite the limitations of our tool, our generated activity diagram, and test cases cover all the paths and transitions covered by the other implementations [7, 38] except for the loops. As shown in Figure 5.2, we achieve 100% test coverage of the activity diagram in terms of activities, branches, conditions, and paths. The activity diagram in [7] contains 10 activities, while our activity diagram contains 12 activities. Since we do not support concurrency, the activity "Eject Card" is repeated to terminate each path, giving us two extra activities. The coverage for transitions is 100% using our tool. We do not consider the additional transitions added by the concurrent steps in [7] because we are achieving the same functionality sequentially using lesser number of transitions. We use corner cases and redundant cases to quantify the quality of the generated test cases. As shown in figure5.5, our tool generates three corner cases and zero redundant cases. Our tool creates more test cases that the other implementations since we have incorporated corner cases. From this preliminary analysis, we can conclude that the tool provides good test coverage and good quality test cases. We have shown that we can generate 1044 test steps in an hour by spending 34 seconds on each test case. We will be using the time, effort and test case productivity parameters in the feasibility study to understand the actual time and effort savings, the test coverage provided and productivity increase that our tool will generate.

5.2 Feasibility Study

The participant pool consisted of ten working professionals with testing knowledge or experience. 50% of the participants had <1 years of software testing experience, 20% had between 1-3 years of experience, and 30% had >3 years of experience. Each participant was given 12 features from 3 prototype applications to be tested manually and automatically. A set of six (numbered from 1-6) features was given to one set of testers for manual test generation, and the same features were given to the other set of testers for automatic test generation. The key parameters that we are interested in are *time taken to generate the test cases*, *number of test cases generated*, *effort to generate the test cases*, *number of test steps*, and *test case productivity*. These parameters are described in section 4.2. All of the 12 features have been tested manually and automatically by 5 participants each. All odd-numbered participants were given features 1-6 to be tested manually and features 7-12 to be tested automatically. All even-numbered participants were given features 7-12 to be tested manually and features 1-6 to be tested automatically.

Feature List	
1. Location	7. Destination
2. Fastest and shortest routes	8. Hazardous condition alert
3. Modify exercise schedule.	9. Add a new trainee
4. Badges earned	10. Recommended exercises and work out history
5. Log into profile	11. Sign up for account
6. Heart rate and stress level	12. Music functionality

Figure 5.6: List of features

5.2.1 Analysis of Quantitative Data

One of the important parameters that we will measure is the time taken to generate test cases. We want to show that the time taken by our tool to generate test cases will be lesser than the time taken using manual methods. Figures 5.7 and 5.8 show the average time taken, the number of test cases, effort, the number of test steps and test case productivity for each user for a set of 6 features.

Manual	User									
	1	2	3	4	5	6	7	8	9	10
Time (s)	105.7	104.7	174.0	312.7	187.7	340.7	119.2	319.7	228.5	102.3
Test cases	2	1	1	5	2	5	2	4	2	1
Effort	51.7	79.7	130.5	63.7	77.2	70.7	67.7	96.0	117.9	102.3
Number of test steps	5	7	5	6	4	6	5	3	5	6
Test case Productivity	329	338	168	317	206	328	273	130	164	192

Figure 5.7: Manually generated test cases - Averages of parameters

Automatic	User									
	1	2	3	4	5	6	7	8	9	10
Time (s)	183.2	198.3	213.5	235.5	156.8	243.0	221.0	291.6	185.0	247.1
Test cases	15	30	9	9	9	5	4	4	3	53
Effort	47.1	12.4	33.5	68.4	31.9	60.1	64.5	146.5	87.2	33.0
Number of test steps	169	314	94	143	92	44	44	26	10	560
Test case Productivity	2699	4893	1291	1633	1889	630	683	363	205	6993

Figure 5.8: Automatically generated test cases - Averages of parameters

Average Time Taken

Overall, when we compare the performance of the participants across all the given features, the automatic method takes more time. The average time taken across all the users per set of features in the manual method is 199.5 seconds, while for automatic it is 217.5 seconds. When we break this down to compare the performance of the same set of features, we find that the average time taken manually for features 1-6 is 162.9 seconds. When the same

set of features 1-6 is tested automatically, the average time is 243.1 seconds. For features 7-12, the average manual time is 236 seconds, while the automatic time is 191.9 seconds. Figure 5.9 compares the performance of each participant for all the features they tested manually and automatically. 5 out of 10 participants took less time with automatic test case generation than manual. These 5 participants are all familiar with the Agile methodology. 3 out of 5 participants who took more time for the automatic method do not have any Agile related experience. Figure 5.10 compares the time taken for each feature manually and automatically.

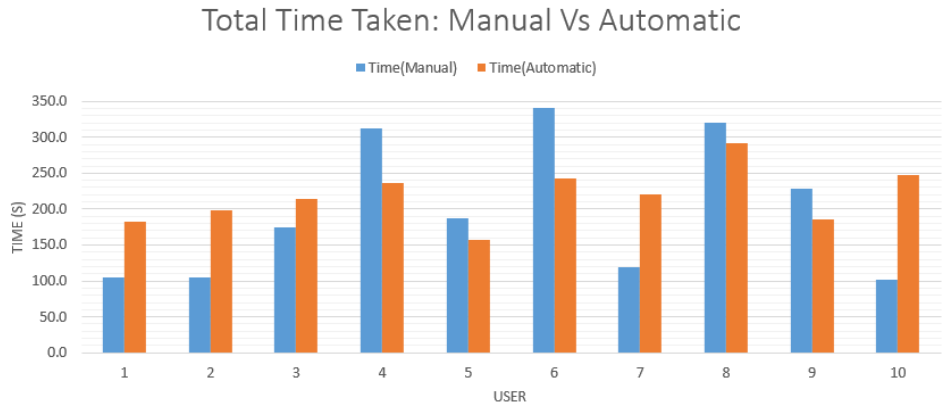


Figure 5.9: Total time taken by each user

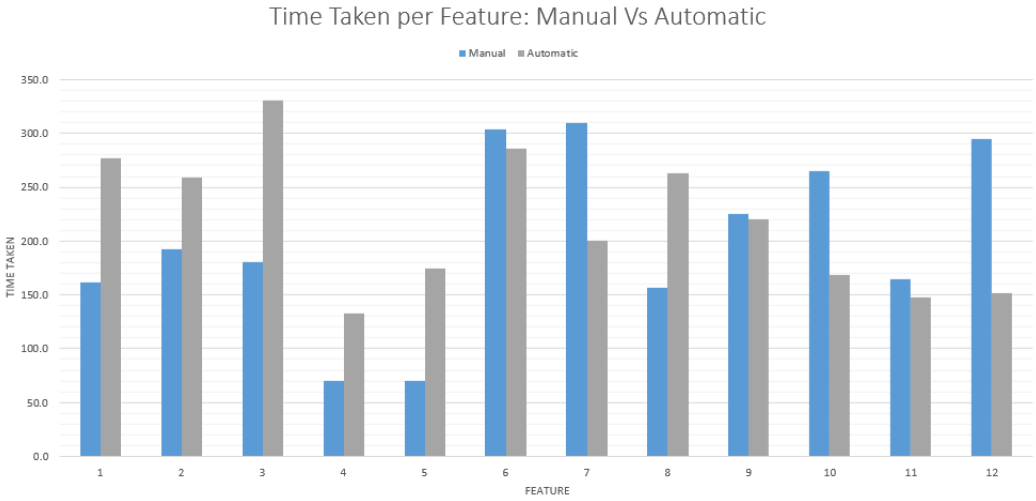


Figure 5.10: Total time taken for each feature

The average time taken for automatic test cases generation by each user comprises of the time required to set up the dictionary, the time required to input the user story, acceptance criteria, and test scenario description, and the generation time. The dictionary set up time is 30% of the total time, and the test scenario takes up 48% of the time. The user story and acceptance criteria input time is 7% and 9% of the total time respectively.

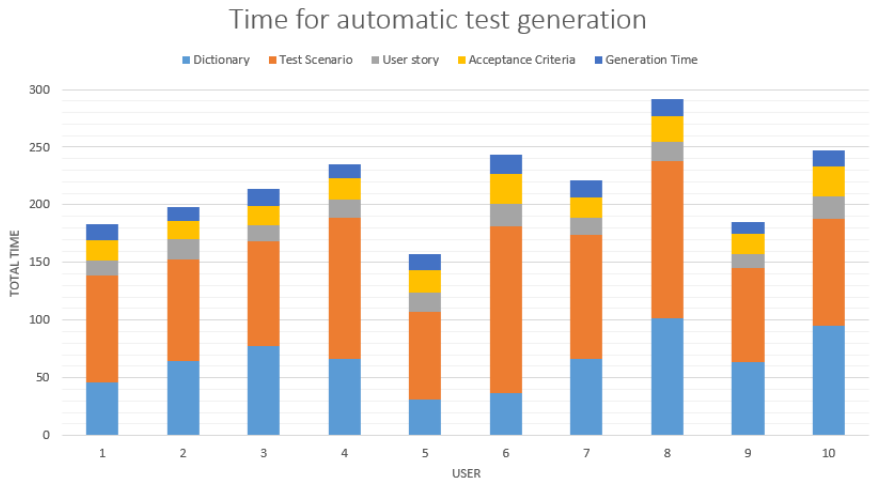


Figure 5.11: Breakdown of time taken in automatic method

From the data above, we can see that on an average the automatic method takes more time than the manual method. However, the number of test cases produced by the automatic method are more than the manual method by a factor of 5.6. Figure 5.12 compares the time taken by the user versus the test cases generated by the user. The number of test cases generated using the automatic method for users 6, 7, 8, and 9 were affected since they did not follow all the guidelines set for the test scenario description. It is observed that even though the automatic method takes more time than the manual method, half the users have performed better using the automatic method with number of test cases greater or equal to the number of test cases generated using the manual method.

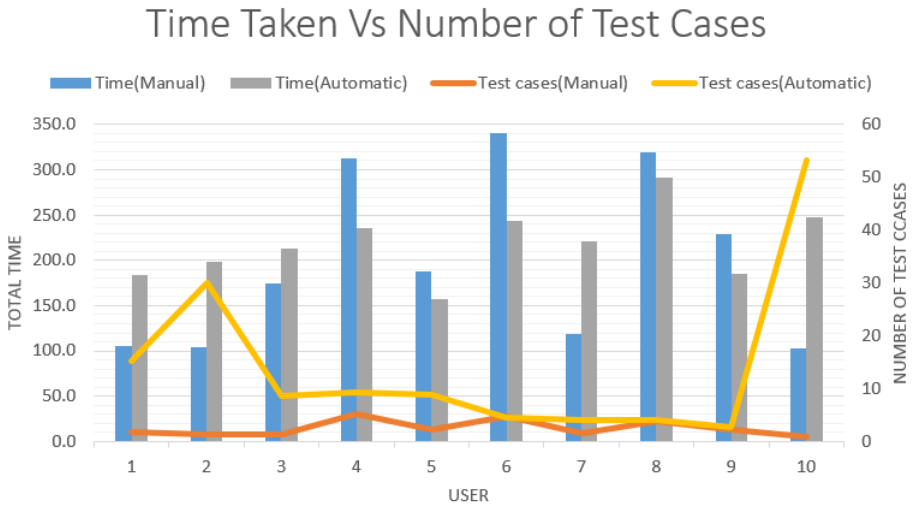


Figure 5.12: Time taken per user Vs number of test cases generated by each user

Effort

Overall, when we compare the performance of the participants across all the given features, the automatic method takes less effort. The average effort taken across all the users per set of features in the manual method is 85.7 seconds per test case, while for automatic it is 58.5 seconds per test case. When we break this down to compare the performance of the same set of features, we find that the average effort for the manual method for features 1-6 is 89 seconds. When the same set of features 1-6 is tested automatically, the average effort is 64 seconds. For features 7-12, the average manual effort is 82.4 seconds, while the automatic effort is 52.8 seconds. Figure 5.13 compares the performance of each participant for all the features they tested manually and automatically. Figure 5.14 compares the effort for each feature manually and automatically. Effort depends on the number of test cases generated.

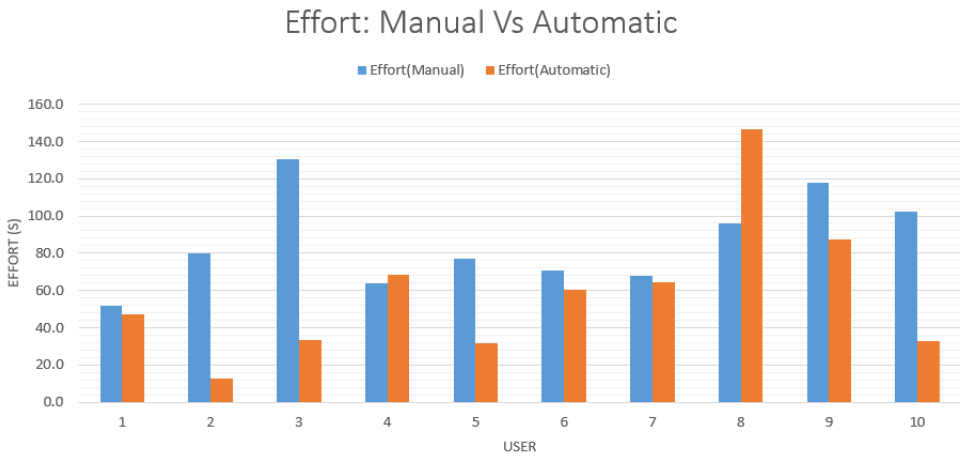


Figure 5.13: Effort taken by each user per test case

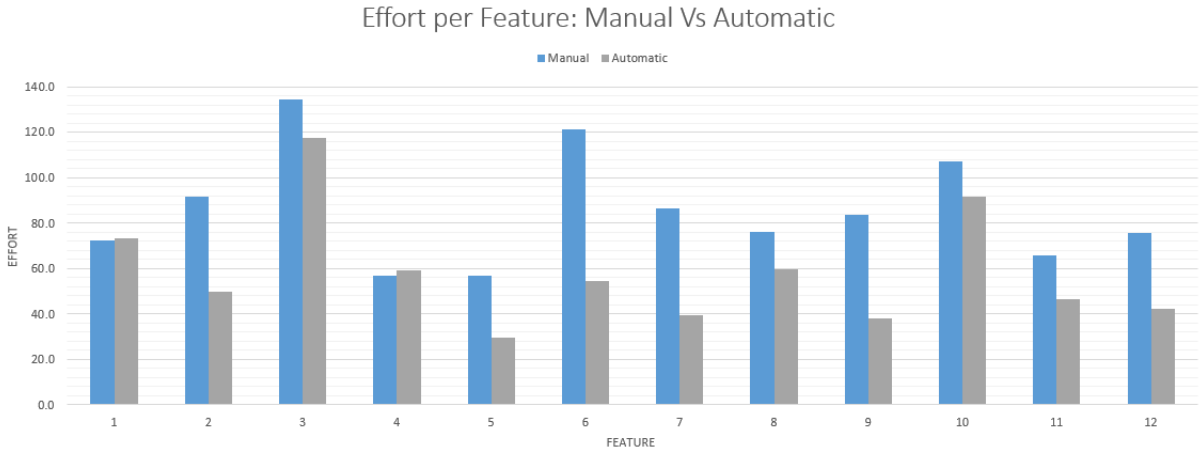


Figure 5.14: Effort taken per feature

Test Case Productivity

Overall, when we compare the performance of the participants across all the given features, the automatic method produces a higher number of test steps which in turn gives a higher TCP. The average TCP taken across all the users per set of features in the manual method is 244 test steps per hour, while for automatic it is 2128 test steps per hour. When we break this down to compare the performance of the same set of features, we find that the average TCP for the manual method for features 1-6 is 228 test steps per hour. When the same set of features 1-6 is tested automatically, the average effort is 2902 test steps per hour. For features 7-12, the average manual effort is 259 test steps per hour, while the automatic effort is 1353 test steps per hour. Figure 5.15 compares the performance of each participant for all the features they tested manually and automatically. Figure 5.16 compares the TCP for each feature manually and automatically.

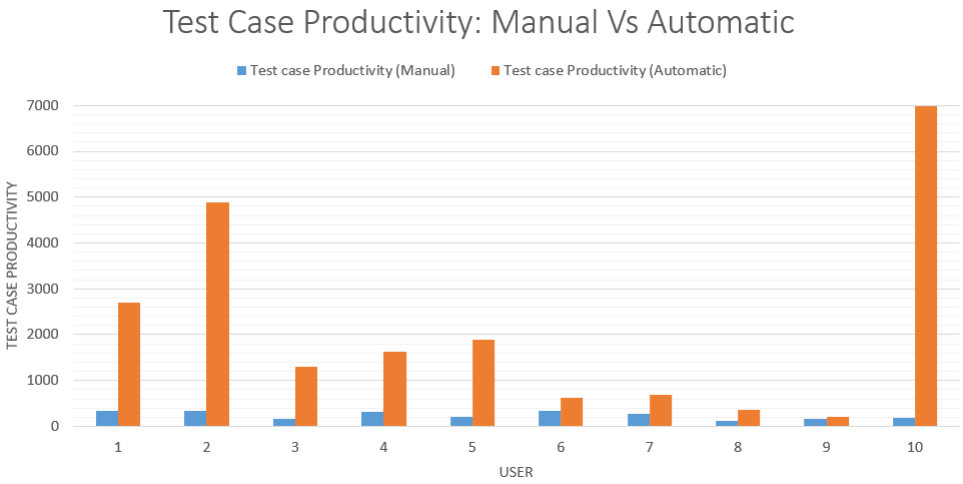


Figure 5.15: TCP by each user

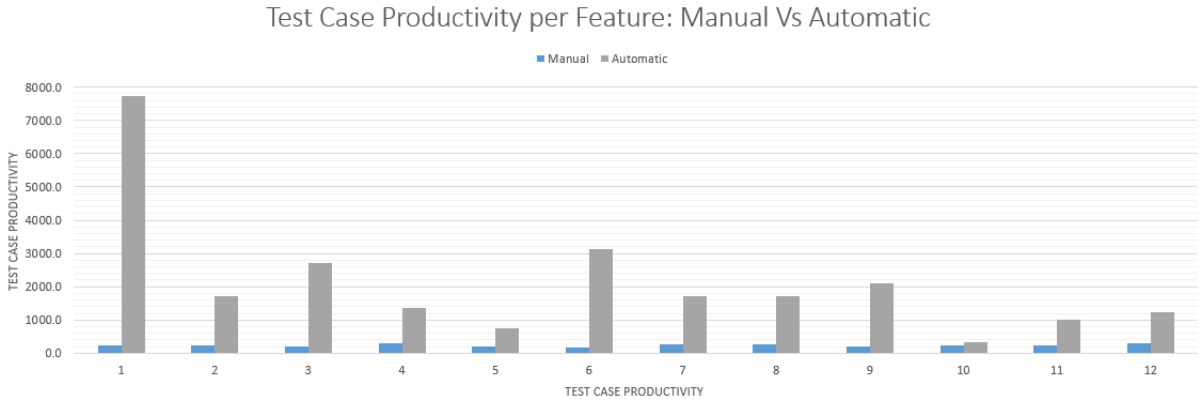


Figure 5.16: TCP per feature

Based on the results from a small subset of 10 users and 12 features, we can see that though the automatic method takes more time as compared to the manual method, it generates more number of test cases and test steps. We want to make sure that the increased number of test cases, test steps, and TCP generated using the automatic method provide more coverage of the functionality than the manual method. We will evaluate the coverage of each of the test cases written by the user manually and generated by the user automatically.

Coverage criteria

As described in Chapter 4, we use activity, path, transition and predicate coverage parameters to verify if all the requirements are met. Figure 5.17 compares the test coverage achieved by each user for all the features. Figure 5.18 compares the test coverage achieved for each feature manually and automatically. The average test coverage across all the users is 57% manually and 69% automatically. The test coverage consists of activity, path, transition and predicate coverage. Using the manual method, the participants achieved 71% activity coverage, 64% transition coverage, 66% path coverage and 28% predicate coverage. Using the automatic method, the participants achieved 75% activity coverage, 70% transition coverage, 78% path coverage and 83% predicate coverage. The test coverage of users 3, 5, 8 and 9 in automatic test case generation was affected because they did not write all the interactions or did not cover all the parallel activities in the functionality.

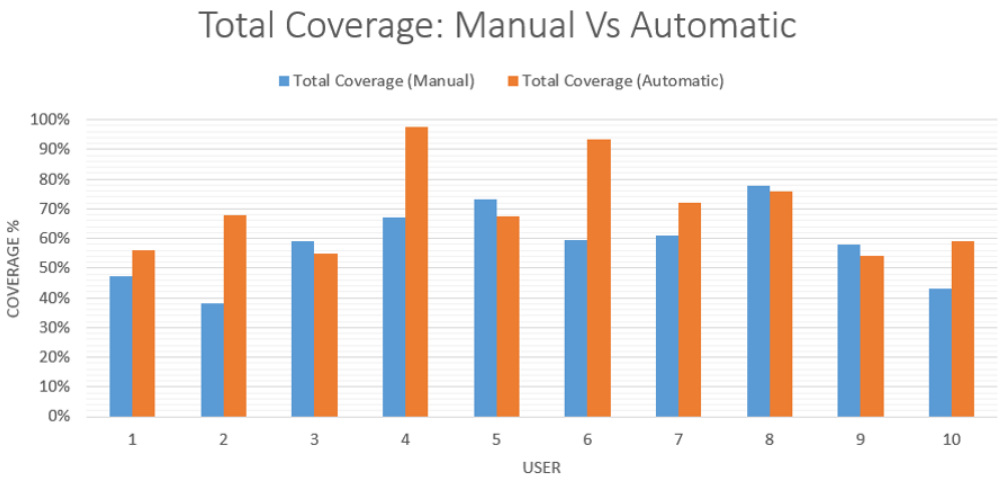


Figure 5.17: Total Coverage by each user

Figure 5.19 and 5.20 show the activity diagrams for feature 3, Modify exercise schedule, generated by user 3 manually and by user 6 automatically. Feature 3 showed an average difference of 13% between the manual and automatic methods. User 3 covered 33% of the requirements manually, while user 6 covered 96% automatically for feature 3.

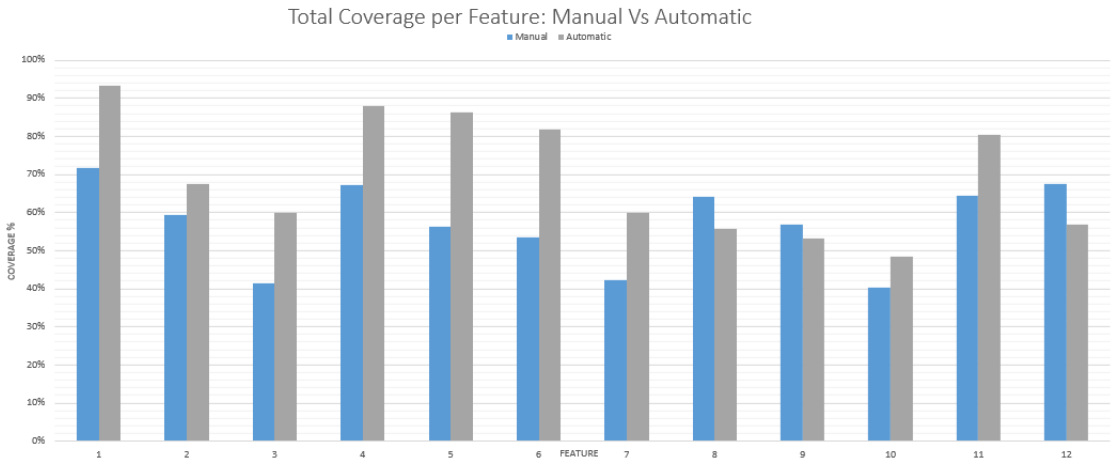


Figure 5.18: Total Coverage per feature

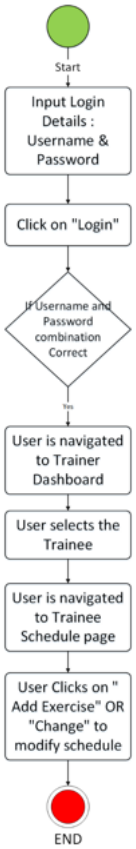


Figure 5.19: Activity diagram for Feature 3 by User 3 (manual)

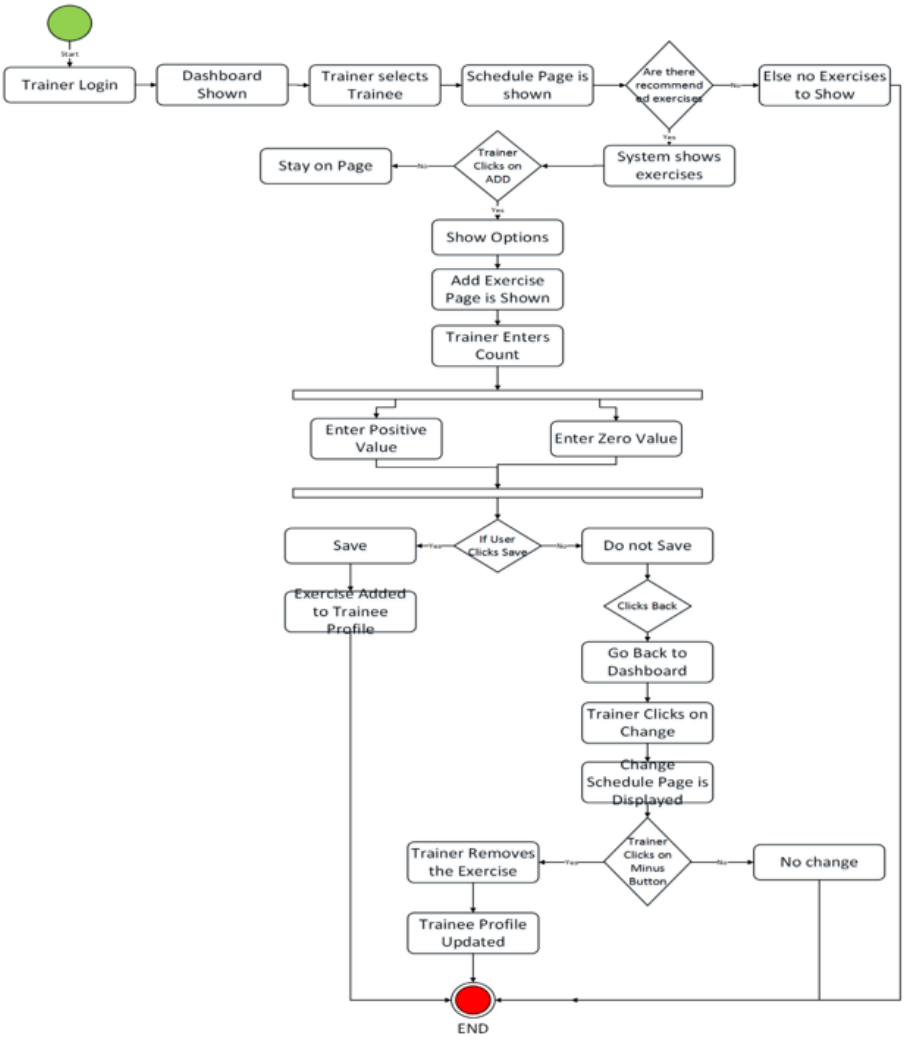


Figure 5.20: Activity diagram for Feature 3 by User 6 (automatic)

Since the TCP using the automatic method is very high compared to the manual method, we analyzed the effect of the test scenario description and dictionary set up time on the number of test steps using correlation and one-way analysis of variance (ANOVA). We analyzed the correlation between the dictionary time, test scenario description time and the number of parameters set up in the dictionary. A high correlation was found between dictionary time and the number of parameters (R = 0.87), but low correlation was observed between test scenario description time and the number of parameters (R = 0.05).

Next, we analyzed the effect of the number of parameters on the number of test steps. A significant effect of the number of parameters was found on the number of steps (p= 0.006) but no significant of test scenario description time was found on the number of test steps (p = 0.39). Hence we can say that for a particular feature if a user sets up a higher number of

	Scenario time	*Dictionary*	*Number of Parameters*	*number of test cases*	*number of test steps*
Scenario time	1				
Dictionary	0.165179299	1			
Number of Parameters	0.057502982	0.870688578	1		
number of test cases	-0.015000584	0.356155342	0.611165342	1	
number of test steps	0.018839769	0.355133423	0.601578902	0.991644469	1

Figure 5.21: Correlation

dictionary parameters then the number of test steps will be high for the feature. Using the TCP formula, the TCP is directly proportional to the number of test steps, hence higher number of dictionary parameters will result in a high TCP.

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Number of Parameters	60	216	3.6	11.63389831
number of test steps	60	8968	149.4666667	169150.4904

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	638313	1	638312.5333	7.54675476	0.006955303	3.921478181
Within Groups	9980565	118	84581.06215			
Total	1.1E+07	119				

Figure 5.22: ANOVA between number of dictionary parameters and number of test steps

Next, we want to test if these results are statistically significant for a larger dataset. We will use hypothesis testing to validate the claims made in Chapter 1.

Hypothesis Testing

The hypotheses that we are trying to prove are the following:

Hypothesis 1: Automatic test case generation using our tool will take less time than manual test case generation.

The null hypothesis is that the average time taken for each feature in automatic test case generation is greater than the average time taken for each feature in manual test case generation.

$$\boxed{\begin{aligned} &\text{H0} = \text{Average T(M)} < \text{Average T(A)} \\ &\text{H1} = \text{Average T(M)} > \text{Average T(A)} \end{aligned}}$$

As shown in the results in Figure 5.23, the P-value is above 0.05. Hence we can say with 95% confidence that we do not have conclusive evidence that the average time for the manual method is less than average time for automatic.

One-Sided t-Test Comparing Time for Manual and Automatic Methods

The TTEST Procedure

Variable: Time

Method	N	Mean	Std Dev	Std Err	Minimum	Maximum
Automatic	12	217.5	64.3213	18.5680	133.1	330.3
Manual	12	199.5	82.8957	23.9299	69.9000	310.2
Diff (1-2)		18.0000	74.1921	30.2888		

Method	Method	Mean	95% CL Mean		Std Dev	95% CL Std Dev	
Automatic		217.5	176.6	258.4	64.3213	45.5649	109.2
Manual		199.5	146.8	252.2	82.8957	58.7229	140.7
Diff (1-2)	Pooled	18.0000	-Infty	70.0102	74.1921	57.3798	105.0
Diff (1-2)	Satterthwaite	18.0000	-Infty	70.1515			

Method	Variances	DF	t Value	Pr < t
Pooled	Equal	22	0.59	0.7208
Satterthwaite	Unequal	20.722	0.59	0.7206

Figure 5.23: Hypothesis testing for time taken

Hypothesis 2: Automatic test case generation using our tool will take less effort than manual test case generation.

The null hypothesis is that the average effort required for each feature in automatic test case generation is greater than the average effort required for each feature in manual test case generation.

$$\boxed{\begin{array}{l} \text{H0} = \text{Average E(M)} < \text{Average E(A)} \\ \text{H1} = \text{Average E(M)} > \text{Average E(A)} \end{array}}$$

As shown in the results in Figure 5.24, the P-value is less than 0.05. Hence we can say that the average effort for automatic is less than the average effort for manual with 95% confidence.

One-Sided t-Test Comparing Effort for Manual and Automatic Methods

The TTEST Procedure

Variable: Effort

Method	N	Mean	Std Dev	Std Err	Minimum	Maximum
Automatic	12	58.4751	25.1374	7.2566	29.6686	117.7
Manual	12	85.7353	24.4482	7.0576	57.0200	134.6
Diff (1-2)		-27.2602	24.7952	10.1226		

Method	Method	Mean	95% CL Mean		Std Dev	95% CL Std Dev	
Automatic		58.4751	42.5035	74.4467	25.1374	17.8072	42.6803
Manual		85.7353	70.2017	101.3	24.4482	17.3190	41.5101
Diff (1-2)	Pooled	-27.2602	-Infty	-9.8783	24.7952	19.1765	35.0939
Diff (1-2)	Satterthwaite	-27.2602	-Infty	-9.8777			

Method	Variances	DF	t Value	Pr < t
Pooled	Equal	22	-2.69	0.0066
Satterthwaite	Unequal	21.983	-2.69	0.0066

Figure 5.24: Hypothesis testing for effort required

Hypothesis 3: Automatic test case generation using our tool will have higher test case productivity than manual test case generation.

The null hypothesis is that the average test case productivity for each feature in automatic test case generation is greater than the test case productivity for each feature in manual test case generation.

$$\begin{aligned} &\text{H0} = \text{Average TCP(M)} > \text{Average TCP(A)} \\ &\text{H1} = \text{Average TCP(M)} < \text{Average TCP(A)} \end{aligned}$$

As shown in the results in Figure 5.25, the P-value is less than 0.05. Hence we can say that the average TCP for automatic is higher than average TCP for manual with 95% confidence.

One-Sided t-Test Comparing Test Case Productivity for Manual and Automatic Methods

The TTEST Procedure

Variable: TCP

Method	N	Mean	Std Dev	Std Err	Minimum	Maximum
Automatic	12	2127.9	1932.1	557.8	322.2	7734.7
Manual	12	244.3	36.9175	10.6572	178.1	300.4
Diff (1-2)		1883.6	1366.5	557.9		

Method	Method	Mean	95% CL Mean		Std Dev	95% CL Std Dev	
Automatic		2127.9	900.3	3355.5	1932.1	1368.7	3280.5
Manual		244.3	220.9	267.8	36.9175	26.1522	62.6814
Diff (1-2)	Pooled	1883.6	925.7	Infty	1366.5	1056.8	1934.0
Diff (1-2)	Satterthwaite	1883.6	881.8	Infty			

Method	Variances	DF	t Value	Pr > t
Pooled	Equal	22	3.38	0.0014
Satterthwaite	Unequal	11.008	3.38	0.0031

Figure 5.25: Hypothesis testing for Test Case Productivity

Hypothesis 4: Automatic test case generation using our tool will have higher test coverage than manual test case generation.

The null hypothesis is that the average test coverage for each feature in automatic test case generation is greater than the average test coverage for each feature in manual test case generation.

$$\boxed{\begin{array}{l} \text{H0} = \text{Average Test Coverage(M)} > \text{Average Test Coverage(A)} \\ \text{H1} = \text{Average Test Coverage(M)} < \text{Average Test Coverage(A)} \end{array}}$$

As shown in the results in Figure 5.26, the P-value is less than 0.05. Hence we can say that the average test coverage of requirements for automatic is higher than average test coverage for manual with 95% confidence.

One-Sided t-Test Comparing Automatic and Manual Test coverage

The TTEST Procedure

Variable: TCoverage (TCoverage)

Method	N	Mean	Std Dev	Std Err	Minimum	Maximum
Automatic	12	0.6929	0.1568	0.0453	0.4833	0.9333
Manual	12	0.5708	0.1084	0.0313	0.4036	0.7167
Diff (1-2)		0.1221	0.1348	0.0550		

Method	Method	Mean	95% CL Mean		Std Dev	95% CL Std Dev	
Automatic		0.6929	0.5933	0.7925	0.1568	0.1111	0.2662
Manual		0.5708	0.5019	0.6397	0.1084	0.0768	0.1840
Diff (1-2)	Pooled	0.1221	0.0276	Infty	0.1348	0.1042	0.1907
Diff (1-2)	Satterthwaite	0.1221	0.0271	Infty			

Method	Variances	DF	t Value	Pr > t
Pooled	Equal	22	2.22	0.0185
Satterthwaite	Unequal	19.561	2.22	0.0192

Figure 5.26: Hypothesis testing for Test Coverage of Requirements

Multiple User stories

In addition to the 12 features listed earlier, we asked the users to generate test cases for 4 smaller functions within the existing functionality. We wanted to compare the time and effort required to generate test cases manually and automatically for these smaller functions.

Sub-feature list
1. View height and weight of trainee
2. View heart rate history
3. Set exercise goals for today
4. Select another song from the queue

Figure 5.27: List of sub-features

As shown in Figure 5.28, the time and effort required for manual generation of test cases are higher than the automatic method. The average time taken using the manual method is 78.2 seconds and 30.2 seconds for automatic. The average effort required for the manual method is 57.1 seconds and 7.38 seconds for automatic. Hence, our tool generates test cases for multiple user stories of a feature, further reducing the time and effort for the tester.

Figure 5.28: Time and effort for sub-features

5.2.2 Analysis of Qualitative Data

The qualitative data in this study is derived from the survey responses and the comments provided by the participants during the study. The survey was conducted throughout the study. Before evaluating the prototype applications, the participants were asked a few descriptive questions about their experience with software testing and Agile software development. Most of the participants were involved in unit and integration testing. 70% of the participants had used the Agile software development methodology. At the end of the study session, the participants were given a survey consisting of nine questions based on their opinion of the tool and their experience with generating test cases manually and automatically for the prototype applications. The first question asked the user to choose between manual and automatic methods to indicate their preference in terms of *better test coverage, time-saving, lesser effort and reusability*. The other eight questions were structured using a Likert scale of 1 to 5, and the participants were asked to provide a reason for their chosen rating. Figure 5.30 lists all the questions posed in the survey and the user's responses to the questions.

Figure 5.29 shows that most of the participants found the automatic method to be the preferred method for generating the test cases in terms of better test coverage, time-saving, lesser effort and reusability. One participant felt that the manual method would be able to generate better test coverage for more complex business scenarios as humans might be able to grasp subtexts and what-if scenarios better than NLP. Another participant stated that manually generating test cases sometimes required less effort. However, the test coverage offered will be poor.

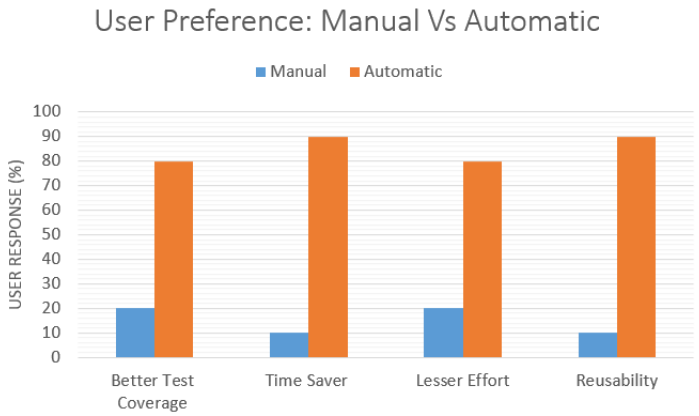


Figure 5.29: User preference - Manual Vs Automatic

	Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful
Rate the usefulness (time vs value) of providing the Test Scenario Description	0	0	1	4	5
Rate the usefulness (time vs value) of the Dictionary feature	0	0	2	1	7
	Not well at all	**Slightly well**	**Moderately well**	**Very well**	**Extremely well**
Rate how well does the Activity Diagram represent the process flow	0	0	2	3	5
	Not accurately at all	**Slightly accurately**	**Moderately accurately**	**Very accurately**	**Extremely accurately**
Rate the Accuracy of the generated test cases	0	0	0	6	4
	Extremely bad	**Somewhat bad**	**Neither good nor bad**	**Somewhat good**	**Extremely good**
Rate the Quality of the generated test cases	0	0	1	3	6
	Extremely easy	**Somewhat easy**	**Neither easy nor difficult**	**Somewhat difficult**	**Extremely difficult**
Select the level of difficulty faced while using the tool	2	3	2	3	0
Rate the ease-of-use of this tool by a user with no testing experience	1	5	2	2	0
	None at all	**A little**	**A moderate amount**	**A lot**	**A great deal**
Based on the testing experience, please rate the relevance of this tool in your work or the industry	1	1	1	4	3

Figure 5.30: Responses to survey questions

Providing the test scenario description is not a part of the existing testing process. It is an extra feature that the user has to provide to generate the test cases. The quality of the test cases is directly dependent on the information provided by the test scenario description. Writing the test scenario description is about 48% of the total time required for automatic

test case generation. We asked the users their opinion on how useful they found providing the test scenario descriptions considering the time investment. 50% of the participants think that it is extremely useful, 40% found it to be very useful. We asked the users the same question about the dictionary feature. 70% of the participants think that the dictionary feature is extremely useful in terms of time versus value. 50% of the participants felt that the activity diagrams represents the process flow extremely well, 30% found it to represent the flow very well. Two participants suggested that concurrent decision making should be incorporated in the future iteration of this tool.

60% of the users felt that the accuracy of the test cases was very accurate and 40% found it to be extremely accurate. Three users responded that provided the test scenario description is in the expected format, the tool generated test case were accurate and covered all the decision flows in the scenario. One participant stated that the accuracy might decrease for complex business scenarios. 60% of the users agreed that quality of the test cases was extremely good, 30% found it to be somewhat good. One participant noted that the test steps sometimes have grammatical errors in them or are cut off. Three participants responded that the quality of the test cases was good since it covered both positive and negative corner cases which the tester may not be able to think of in manual testing. Another participant suggested that for better results the tool could have some checks on the validity/accuracy of test cases.

30% of the users found the tool to be somewhat difficult to use, and 30% found it somewhat easy to use. One participant responded that if the test scenario description is not entered in the required format, the tool does not work accurately. Hence the format has to be simplified. Two users felt the tool has a steep learning curve. The users were asked if based on their testing experience, they found the tool to be relevant in their work or industry. 30% rated the relevance as a great deal, and 40% rated it as a lot. Two participants think that the tool would be very relevant in user interface testing. One participant responded none at all to this question since the testing in his company is based on acceptance criteria, and they do not write test cases. One participant responded that it is easy for test developers to miss out on corner cases or testing functionality based on real user scenarios and this tool can help test developers identify gaps in their test cases and significantly improve the quality of the end product. Another participant felt that the relevance of the tool would be higher if the natural language processing were expanded to understanding context and semantics. 50% of the participants think that the tool is somewhat easy-to-use by a user with no testing background. Three participants found the tool to be user-friendly and intuitive. 20% think that it would be somewhat difficult for a new user to think from a tester's perspective to write a good test scenario description in the correct format. But once the user is acquainted with the tool, the overall value is greater than the time invested in learning the format.

5.2.3 Discussion of Results

In this section, we will discuss the results presented in Sections 5.2.1 and 5.2.2. We will summarize the trends observed in the quantitative data obtained during manual and automatic test case generation.

For the feasibility study, the time taken to generate the test cases was used to measure which method is faster, manual or automatic. The effort spent on generating test cases is the average time that a tester spends on creating each test case. In comparison with manual test case generation, the automatic test case generation tool increases the time required to generate test cases by 30% but decreases the effort per test case by 31% and increases the TCP by 827%. When integrated into the Agile testing process, the user story and acceptance criteria input time can be eliminated because they are provided to the testers and can be pre-populated in the tool. Elimination of user story and acceptance criteria input time changes the increase in time to 7%. The total coverage of requirements was improved by 23% using automatic test generation. The automatic method improved the activity coverage by 7%, transition coverage by 10%, path coverage by 18% and predicate coverage by 83%.

For the tool to work efficiently, it is important that the user enters consistent and relevant information in the test scenario description and the dictionary. The tool will not retrieve boundary conditions even if they are set up if the user does not include the key parameters in the test scenario description. It was observed that two participants missed some information in the test scenario and did not include the dictionary parameter in the test scenario description. Hence their test cases, test steps, effort, and test case productivity were impacted negatively. This holds true for sub-features as well.

Using hypothesis testing we could not prove that automatic test case generation takes less time than manual test case generation. But there is still not enough evidence to say that manual test case generation takes less time as compared to the automatic method. The participants did not have any previous experience with the tool. They were given a 10-minute training session with the tool, before starting the study. Since half the participants took less time using the automatic method, Hypothesis 1 should be revisited after the participants are familiar with the tool. We showed that the effort required for automatic test case generation is less than the effort required manually. We also showed that the test case productivity and test coverage is higher for the automatic method.

5.3 Findings of the Preliminary Analysis & Feasibility Study

We will conclude the discussion by summarizing our findings and validating our claims made in Section 1.5, Contributions. We had claimed that the tool could be used by testers who

may or may not be experienced, to achieve improvement in testing within Agile software development. The tool did not reduce the time taken to create test cases but increased it by 7%. We expect the time taken to reduce over time when the participants get familiar with the tool. The tool reduced the effort required to create test cases by 31%. The tool improved the quality of test cases by incorporating corner cases. There was a 93% increase in corner cases as compared to the manual method. The tool improved the test coverage of the requirements by 23%. The tool can also generate test cases for multiple user stories of a feature thus further reducing the time and effort for the tester by 61% and 87% respectively.

From the results of the preliminary study, we can see that our tool provides 100% test coverage of activities, paths, and transitions. The activity diagram is generated by the tool using NLP techniques. From the results of the feasibility study, we can verify that our tool produces a higher number of test cases in lesser time. Test cases with less redundancy and with test steps that cover the boundary conditions improve the quality of the test. As shown in the results our tool provides corner cases using the Dictionary feature. The number of redundant cases is also lower. The number of test steps created per test cases is higher using automatic test creation which results in higher test case productivity. Test cases can be generated for multiple user stories using the tool by spending lesser time and effort.

Chapter 6

Conclusion and Future Work

6.1 Contributions of Automatic Test Case Generation Tool

In this thesis, we designed and implemented a tool to support the auto-generation of test cases within an Agile software development workflow using natural language processing. This tool increases the time taken to generate the test cases by 7% but reduces the effort required to generate the test cases by 31%. The participants are accustomed to writing the test cases manually, but they had no prior experience with the tool. Hence, we would expect the time spent on generating the test cases using the tool to reduce over time. The test coverage of the requirements is improved by 23%. The quality of the test cases represented by the number of redundant cases and corner cases is high. The time and effort for generating test cases for multiple user stories of a feature are reduced by 61% and 87% respectively. The work in the thesis is the first attempt at replacing manual test case creation in the Agile process. It would need several iterations before it turns into a final product. In conclusion, we can generate test cases from user stories in an Agile software development workflow while reducing the effort for testers and improving the test coverage of requirements.

6.2 Future Work for Automatic Test Case Generation

There are several directions in which this work can be expanded and enhanced. First, the activity diagrams can be improved to support parallel flows of activities or concurrent activities and looping back to previous activities. The process of writing the test scenario description will simplify if we can provide multiple IF-ELSE constructs to support concurrency and looping. This would require supporting elements such as fork, join, and merge.

Second, the dependency tags in the frame structures for active/passive voice, prepositions, preconditions and other parts of speech must be updated with the help of a linguistic expert. In the current form, the frames do not always generate grammatically correct sentences.

Third, the performance and utility of this tool could be vastly improved if it supported understanding of context or semantics. The user would not be restricted to using consistent language in the user story, dictionary and test scenario description. The quality of the test cases would improve if the tool could understand the meaning of the test steps in the dictionary.

Fourth, Agile Model Driven Development (AMDD) is an upcoming Agile version of the model-driven software development approach. In AMDD, requirements are not gathered through detailed specifications but instead in the form of free-form diagrams using white-boards and paper. A combination of Forsyth's [6] storyboarding tool and our test cases generation tool would be a good solution for implementing the AMDD methodology.

Fifth, the tool could be integrated into test management software such as Zephyr to provide a complete testing solution for Agile software development.

Finally, in future iterations of the feasibility study, the user story and acceptance criteria can be pre-populated reducing the automatic generation time. It would also be helpful to give users more time before running the experiment to become acquainted with the correct way of writing the test scenario description.

Bibliography

[1] S. W. Ambler, *The object primer: Agile model-driven development with UML 2.0*. Cambridge University Press, 2004.

[2] B. K. Aichernig, F. Lorber, and S. Tiran, "Formal test-driven development with verified test cases," in *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pp. 626–635, Jan 2014.

[3] D. North *et al.*, "Introducing bdd," *Better Software, March*, 2006.

[4] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 383–387, Aug 2011.

[5] X. Fan, J. Shu, L. Liu, and Q. Liang, "Test case generation from uml subactivity and activity diagram," in *Electronic Commerce and Security, 2009. ISECS '09. Second International Symposium on*, vol. 2, pp. 244–248, May 2009.

[6] J. B. Forsyth, "Exploring electronic storyboards as interdisciplinary design tools for pervasive computing," 2015.

[7] A. K. Jena, S. K. Swain, and D. P. Mohapatra, "A novel approach for test case generation from uml activity diagram," in *Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on*, pp. 621–629, Feb 2014.

[8] P. Kulkarni and Y. Joglekar, "Generating and analyzing test cases from software requirements using nlp and hadoop," *International Journal of Current Engineering and Technology (INPRESSCO)*, 2014.

[9] R. van den Broek, M. M. Bonsangue, M. Chaudron, and H. van Merode, "Integrating testing into agile software development processes," in *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pp. 561–574, IEEE, 2014.

[10] "Practical approach in creating agile test cases, https://josephvargheese.wordpress.com/2012/11/04/practical-approach-in-creating-agile-test-cases/," Jan 2014.

[11] P. VII, *Agile Product Management: User Stories: How to Capture Requirements for Agile Product Management and Business Analysis with Scrum.* Pashun Consulting Ltd.

[12] A. Van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in *Proceedings of the 22nd international conference on Software engineering*, pp. 5–19, ACM, 2000.

[13] Z. Ahmad, M. Hussain, A. Rehman, U. Qamar, and M. Afzal, "Impact minimization of requirements change in software project through requirements classification," in *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, p. 15, ACM, 2015.

[14] "Manage agile testing: Easy test management for qa teams, https://www.versionone.com/product/lifecycle/test-management/."

[15] "Test management add-ons for atlassian - zephyr, https://www.getzephyr.com/products/zephyr-for-jira."

[16] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based automated black-box test generation," in *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pp. 489–495, IEEE, 2001.

[17] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, (New York, NY, USA), pp. 31–36, ACM, 2007.

[18] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan, "A survey on automatic test case generation," *Academic Open Internet Journal*, vol. 15, no. 6, 2005.

[19] P. N. Boghdady, N. L. Badr, M. A. Hashim, and M. F. Tolba, "An enhanced test case generation technique based on activity diagrams," in *Computer Engineering & Systems (ICCES), 2011 International Conference on*, pp. 289–294, IEEE, 2011.

[20] N. Ismail, R. Ibrahim, *et al.*, "Automatic generation of test cases from use-case diagram," in *Proceedings of the International Conference on Electrical Engineering and informatics Institut Teknologi Bandung*, Institut Teknologi bandung, 2007.

[21] J. Ryser and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts," in *Proc. 12th International Conference on Software and Systems Engineering and their Applications*, Citeseer, 1999.

[22] E. G. Cartaxo, F. G. Neto, and P. D. Machado, "Test case generation by means of uml sequence diagrams and labeled transition systems," in *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pp. 1292–1297, IEEE, 2007.

[23] R. Sharma, S. Gulia, and K. Biswas, "Automated generation of activity and sequence diagrams from natural language requirements," in *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*, pp. 1–9, IEEE, 2014.

[24] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit.," in *ACL (System Demonstrations)*, pp. 55–60, 2014.

[25] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pp. 423–430, Association for Computational Linguistics, 2003.

[26] M.-C. De Marneffe, B. MacCartney, C. D. Manning, *et al.*, "Generating typed dependency parses from phrase structure parses," in *Proceedings of LREC*, vol. 6, pp. 449–454, 2006.

[27] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," tech. rep., Technical report, Stanford University, 2008.

[28] "Universal dependencies, http://universaldependencies.org/introduction.html."

[29] M.-C. De Marneffe, T. Dozat, N. Silveira, K. Haverinen, F. Ginter, J. Nivre, and C. D. Manning, "Universal stanford dependencies: A cross-linguistic typology.," in *LREC*, vol. 14, pp. 4585–92, 2014.

[30] S. Schuster and C. D. Manning, "Enhanced english universal dependencies: An improved representation for natural language understanding tasks," in *Proceedings of the 10th International Conference on Language Resources and Evaluation*, 2016.

[31] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[32] B. Santorini, "Part-of-speech tagging guidelines for the penn treebank project (3rd revision)," 1990.

[33] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[34] G. Miller and C. Fellbaum, "Wordnet: An electronic lexical database," 1998.

[35] J. Bhatia, R. Sharma, K. K. Biswas, and S. Ghaisas, "Using grammatical knowledge patterns for structuring requirements specifications," in *2013 3rd International Workshop on Requirements Patterns (RePa)*, pp. 31–34, July 2013.

[36] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from uml activity diagram based on gray-box method," in *Software Engineering Conference, 2004. 11th Asia-Pacific*, pp. 284–291, IEEE, 2004.

[37] D. Kundu and D. Samanta, "A novel approach to generate test cases from uml activity diagrams.," *Journal of Object Technology*, vol. 8, no. 3, pp. 65–83, 2009.

[38] O. Oluwagbemi and A. Hishammuddin, "Automatic generation of test cases from activity diagrams for uml based testing (ubt)," *Jurnal Teknologi*, vol. 77, p. 13, 2015.

[39] R. Elghondakly, S. Moussa, and N. Badr, "Waterfall and agile requirements-based model for automated test cases generation," in *Intelligent Computing and Information Systems (ICICIS), 2015 IEEE Seventh International Conference on*, pp. 607–612, IEEE, 2015.

[40] S. Shukla and G. Chandel, "A systematic approach for generate test cases using uml activity diagrams," *International Journal of Research in Management and Technology*, vol. 2, pp. 469–475, 2012.

[41] M. A. Finlayson, "Java libraries for accessing the princeton wordnet: Comparison and evaluation," in *Proceedings of the 7th Global Wordnet Conference, Tartu, Estonia*, 2014.

[42] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pp. 173–180, Association for Computational Linguistics, 2003.

[43] I. S. Bajwa, M. Lee, and B. Bordbar, "Translating natural language constraints to ocl," *Journal of King Saud University-Computer and Information Sciences*, vol. 24, no. 2, pp. 117–128, 2012.

[44] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for uml activity diagrams," in *Proceedings of the 2006 international workshop on Automation of software test*, pp. 2–8, ACM, 2006.

[45] L. Gulechha, "Software testing metrics,"

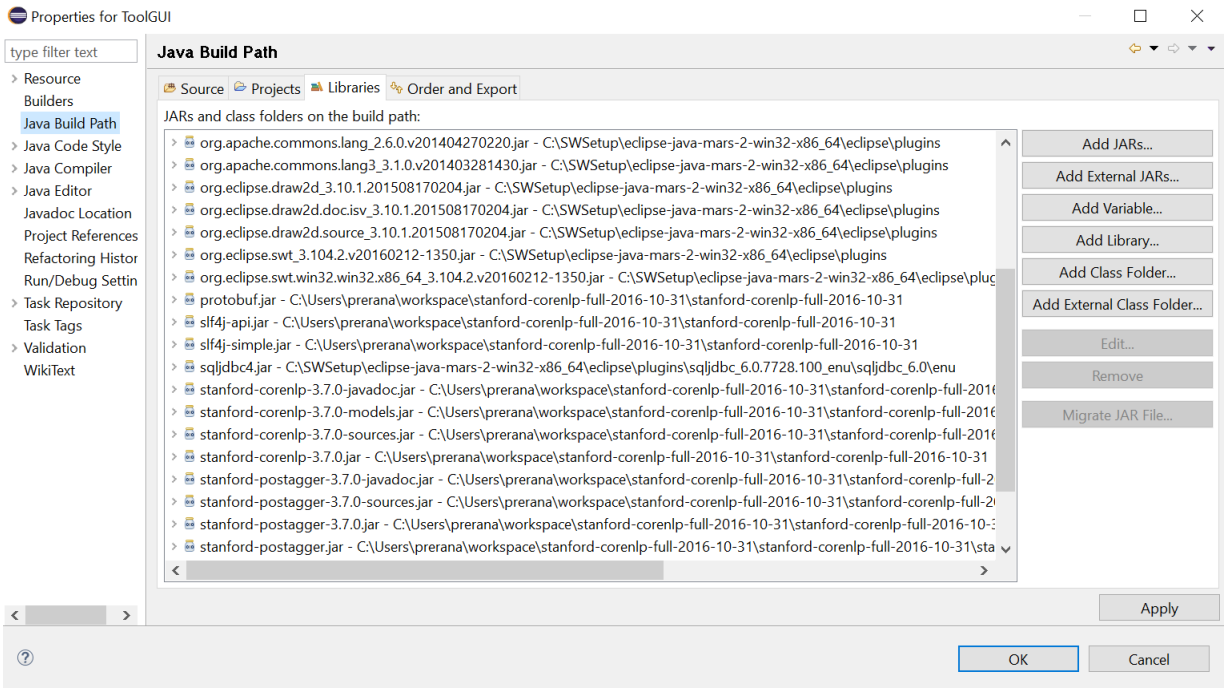
Appendix A

Installation Manual

The following software are required to run the Test Case Generation Tool.

1. Install Eclipse Mars IDE. Import the additional packages: Draw2D (org.eclipse.draw2d), java.awt, java.swing, org.eclipse.swt and org.eclipse.swt.widgets.

2. Install the Stanford Core NLP (http://stanfordnlp.github.io/CoreNLP/) and Stanford POS Tagger. Import all the Executable JAR files into your Java project in Eclipse.



3. Install the SQL Server 2016 to setup the database. Create a table using the command "CREATE TABLE DEMO(SCENARIO TEXT)".

4. Install Wordnet 3.0 (https://wordnet.princeton.edu/wordnet/download/).
5. Install JWI (http://projects.csail.mit.edu/jwi/)

The main files in the project are:

ToolGUI.java - This file contains the main to run the GUI. The code for creating the GUI, inputting the data entered by the user and searching the database (test scenario descriptions) for matching user stories is located in this file.

DBRetrieve.java - This file contains the code to retrieve test scenarios from the database and display in a JTable.

Parser.java - Implements the Stanford Tagger and Parser to get the tags and dependencies. This information is used to create frame data

StanfordLemmatizer.java - This file contains the code to get the lemma and synonyms of words.

StringMatcher.java - Searches for the user story (verbs, nouns and synonyms) in the test scenario description and returns the line where match is found.

Dictionary.java - Implements the Dictionary functionality.

ActDiag.java - This file contains the code for creating the activity diagram from the frame structure.

ActGraph.java - Contains the code for creating the activity graph from frames. The activities are matched with parameters in the dictionary to retrieve test steps. Test steps from the dictionary are added to the graph. The graph serves as an input to the DFS algorithm. Test cases are generated and populated in the GUI.

Appendix B

IRB Approval

This appendix contains the IRB Approval for the feasibility study.

VirginiaTech

Office of Research Compliance
Institutional Review Board
North End Center, Suite 4120, Virginia Tech
300 Turner Street NW
Blacksburg, Virginia 24061
540/231-4606 Fax 540/231-0959
email irb@vt.edu
website http://www.irb.vt.edu

MEMORANDUM

DATE: February 14, 2017

TO: Thomas L Martin, Prerana Pradeepkumar Rane

FROM: Virginia Tech Institutional Review Board (FWA00000572, expires January 29, 2021)

PROTOCOL TITLE: Automatic Generation of Test Cases for Agile using Natural Language Processing

IRB NUMBER: 17-091

Effective February 14, 2017, the Virginia Tech Institution Review Board (IRB) Chair, David M Moore, approved the New Application request for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report within 5 business days to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at:

http://www.irb.vt.edu/pages/responsibilities.htm

(Please review responsibilities before the commencement of your research.)

PROTOCOL INFORMATION:

Approved As:	**Expedited, under 45 CFR 46.110 category(ies) 5,7**
Protocol Approval Date:	**February 14, 2017**
Protocol Expiration Date:	**February 13, 2018**
Continuing Review Due Date*:	**January 30, 2018**

*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

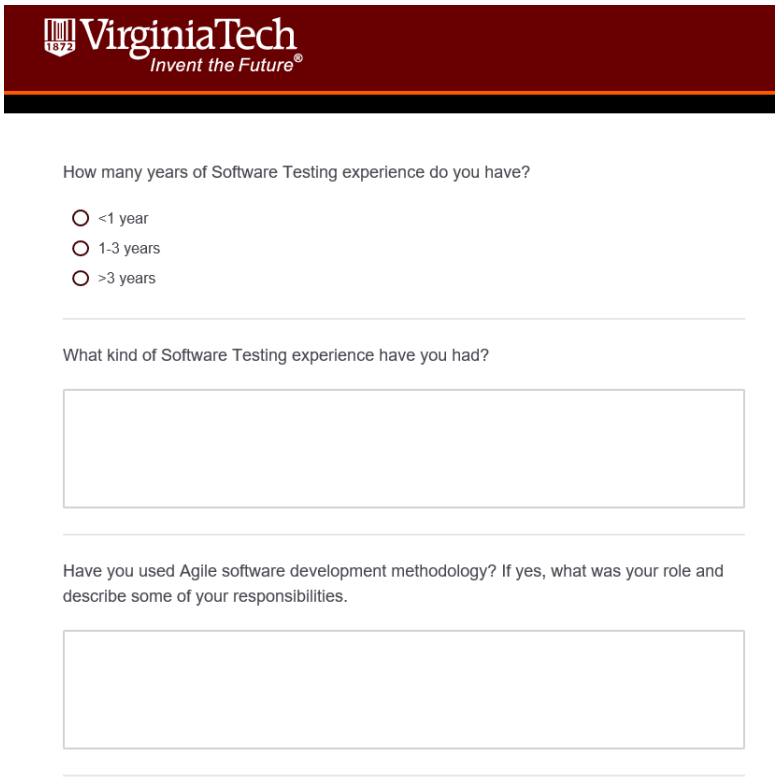
FEDERALLY FUNDED RESEARCH REQUIREMENTS:

Per federal regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals/work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Appendix C

Survey for Feasibility Study



Data for manually generated test cases.

	Time taken (s)	Number of test cases	Number of test steps/test case	Corner cases covered?
App 1 Feature 1				
App 1 Feature 2				
App 2 Feature 1				
App 2 Feature 2				
App 3 Feature 1				
App 3 Feature 2				
App X Sub-Feature 1.1				
App X Sub-Feature 1.2				

Data for automatically generated test cases using the tool.

	Time taken (s)	Number of test cases	Number of test steps/test case	Corner cases covered?	Redundant Test cases
App 1 Feature 1					
App 1 Feature 2					
App 2 Feature 1					
App 2 Feature 2					
App 3 Feature 1					
App 3 Feature 2					
App X Sub-Feature 1.1					
App X Sub-Feature 1.2					

Please indicate your preferred method for the following criteria.

	Manual	Automatic
Better Test Coverage	○	○
Time Saver	○	○
Lesser Effort	○	○
Reusability	○	○

Please provide reasons for your choices above.

Rate the usefulness (time vs value) of providing the Test Scenario Description?

Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful
○	○	○	○	○

Please provide a reason for your rating above.

Rate the usefulness (time vs value) of the Dictionary feature?

Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful
○	○	○	○	○

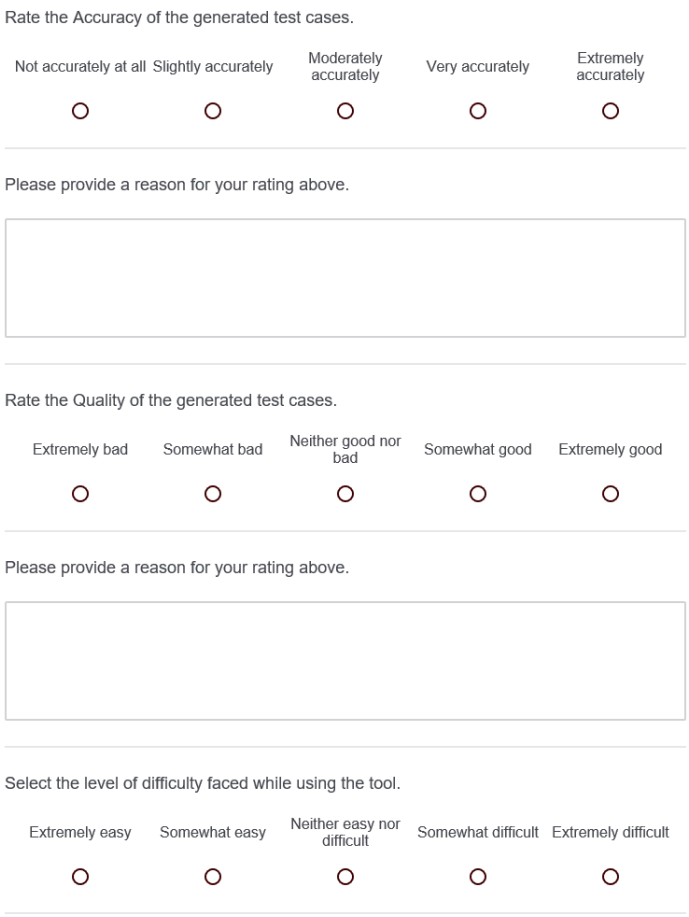
Please provide a reason for your rating above.

Rate how well does the Activity Diagram represent the process flow?

Not well at all	Slightly well	Moderately well	Very well	Extremely well
○	○	○	○	○

Please provide a reason for your rating above.

In your opinion, can this tool be used for improving the requirement gathering process (modeling changes in requirements/ offering better alternatives)?


Rate the Accuracy of the generated test cases.
Not accurately at all
Slightly accurately
Moderately accurately
Very accurately
Extremely accurately
Please provide a reason for your rating above.
Rate the Quality of the generated test cases.
Extremely bad
Somewhat bad
Neither good nor bad
Somewhat good
Extremely good
Please provide a reason for your rating above.
Select the level of difficulty faced while using the tool.
Extremely easy
Somewhat easy
Neither easy nor difficult
Somewhat difficult
Extremely difficult

Please provide a reason for your rating above.

Based on the testing experience, please rate the relevance of this tool in your work or the industry.

None at all	A little	A moderate amount	A lot	A great deal
○	○	○	○	○

Please provide a reason for your rating above.

Rate the ease-of-use of this tool by a user with no testing experience.

Extremely easy	Somewhat easy	Neither easy nor difficult	Somewhat difficult	Extremely difficult
○	○	○	○	○

Please provide a reason for your rating above.

Please provide comments, suggestions or improvements if any.

Appendix D

Data sheet for test cases generated manually and automatically

Data from test cases generated manually and automatically

Feature	Manual Test Cases	Data entered by user in the tool	Test cases from tool
App 1 AstroNav: AstroNav helps astronauts on a planetary surface navigate their way to a destination.			
1. Get current location		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
2. Set desired destination		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
3. View fastest (time) and shortest (distance) route		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
4. Hazardous condition alert		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
App 2 FitTrain: FitTrain is an app in which trainers set exercises for the users. Users can log into their profile and select exercises for today from a list of exercises.			
1. Modify exercise schedule.		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
2. Add a new trainee Sub feature: View height and weight of user		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
3. See my badges		User story: Acceptance Criteria:	

		Test Scenario Description: Dictionary:	
4. View user's workout history and recommend exercises. Sub feature: Set exercise goal for today		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
App 3: Mooditation is an app to monitor your stress levels using Heart Rate and to play music based on your stress levels to calm you down.			
1. Log into profile		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
2. Signing up for a new account		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
3. View heart rate for 7 weeks and stress level for today Sub feature : view heart rate history for 7 weeks		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	
4. Listen to music based on stress level Sub feature: Select another song from the queue		User story: Acceptance Criteria: Test Scenario Description: Dictionary:	